GPTune: Multitask Learning for Autotuning Exascale Applications

Yang Liu
Berkeley Laboratory
USA
liuyangzhuan@lbl.gov

Wissam M. Sid-Lakhdar
Berkeley Laboratory
USA
wissam.sidlakhdar@gmail.com

Osni Marques
Berkeley Laboratory
USA
oamarques@lbl.gov

Xinran Zhu
Cornell University
USA
xz584@cornell.edu

Chang Meng
Emory University
USA
chang.meng@emory.edu

James W. Demmel
University of California, Berkeley
USA
demmel@cs.berkeley.edu

Xiaoye S. Li
Berkeley Laboratory
USA
xsli@lbl.gov

Abstract

Multitask learning has proven to be useful in the field of machine learning when additional knowledge is available to help a prediction task. We adapt this paradigm to develop autotuning frameworks, where the objective is to find the optimal performance parameters of an application code that is treated as a black-box function. Furthermore, we combine multitask learning with multi-objective tuning and incorporation of coarse performance models to enhance the tuning capability. The proposed framework is parallelized and applicable to any application, particularly exascale applications with a small number of function evaluations. Compared with other state-of-the-art single-task learning frameworks, the proposed framework attains up to 2.8X better code performance for at least 80% of all tasks using up to 2048 cores.

CCS Concepts • **Mathematics of computing** → **Probability and statistics**; • **Computing methodologies** → **Machine learning**; *Distributed computing methodologies.*

Keywords autotuning, multitask learning, machine learning, Bayesian optimization, Exascale Computing Project

1 Introduction

In preparation for the upcoming era of exascale computing, significant effort is being invested to develop highly scalable numerical libraries and high-fidelity modeling and simulation codes across the spectrum of science and engineering domains and disciplines. Achieving optimal performance and performance portability of all these codes has become an increasingly intractable problem. At least two challenges must be addressed. The first challenge is due to diverse exascale computer architectures with heterogeneous nodes, deep and complex memory hierarchies and varying interconnect speeds. The traditional latency-bandwidth model of parallel computing is far from adequate for parallel runtime prediction. The second challenge lies in the application codes themselves. Most of the exascale application codes have a number of tunable parameters that affect performance, and oftentimes the simulations involve expensive "function evaluations", requiring either long runtime or many hardware resources (e.g., core count), so that the brute-force "grid search" approach to find optimal parameters is infeasible. There is an increasing demand for performance autotuning tools, or autotuners, that can automate the tuning process. Autotuning strategies are concerned with automatically picking the right set of parameters to optimally solve a particular problem on a given architecture. For exascale applications with a small number of allowed runs, the critical metrics for the autotuners are final performance, i.e., the best possible performance after the tuner finishes, and anytime performance, i.e., the best performance so-far when tuning is terminated early.

PPoPP '21, February 27–March 3, 2021, Virtual Event, Republic of Korea
© 2021 Association for Computing Machinery.
ACM ISBN 978-1-4503-8294-6/21/02…$15.00
https://doi.org/10.1145/3437801.3441621

We have been developing a publicly available autotuning framework called GPTune using statistical and machine learning techniques. Our goals are: 1) Support a variety of measurable performance metrics, including runtime, memory usage, accuracy, energy, or their hybrids like minimizing time given a memory constraint; 2) Support distributed-memory MPI codes running on large-scale machines; 3) Support archiving and reusing tuning data from multiple executions to allow tuning to improve over time; 4) Allow incorporation of new optimization techniques to improve the tuning process.

Our main contributions in GPTune and this paper are summarized below.

- The core idea is to introduce *multitask* learning [24, 32] in the Gaussian processes (GP) in Bayesian optimization. Thus, we can simultaneously tune multiple problem instances of a given application instead of one at a time. This is particularly attractive for exascale applications, because it greatly reduces the number of function evaluations compared to single-task learning.
- GPTune allows the application code to be run on any type of parallel machine. Moreover, we parallelized GPTune itself for distributed memory platforms using MPI, so that the tuning process is fast and scalable.
- In addition to standard tuning needs, GPTune includes more advanced features, such as multi-objective tuning, and can take advantage of coarse performance models provided by the users.
- We have evaluated GPTune's functionalities for a variety of MPI based applications ranging from mathematical libraries to production-level scientific simulation codes. Experimental results show that GPTune significantly outperforms single-task learning and two other state-of-the-art methods in terms of both final and anytime performance, particularly when the budget (number of allowed runs of the application) is low.

The remainder of this paper is organized as follows. Section 2 defines the terms used in the paper. Section 3 describes the multitask learning algorithms used in GPTune. Section 4 describes the programming model used in the parallel implementation of GPTune. Section 5 describes the related work in autotuning and black-box optimization for autotuning. Section 6 shows the autotuning results with several HPC math libraries and application codes.

2 GPTune: Definitions and Notations

GPTune uses Bayesian optimization to iteratively build a GP surrogate model, by running the application at a few carefully chosen tuning parameter values. Instead of the standard GP method, GPTune relies on Multitask Learning Autotuning (MLA). MLA means using performance data from multiple tasks (e.g. a fixed linear algebra operation on k matrices of several dimensions $t_1, t_2, \ldots, t_k$) to build a joint surrogate model $f(t, x)$ of the true runtime $y(t, x)$ to use for tuning. When the performance varies reasonably smoothly as a function of t, using all the available data to build $f(t, x)$ can make it more accurate than the surrogate model for each individual task.

In our notation, the phrase "task parameter" will refer to an input, like t above, that defines the task to be solved; there are generally multiple task parameters needed to define a task. The phrase "tuning parameter" will refer to a parameter to be tuned, like x above; again there are generally multiple parameters for one task. The phrase "parameter configuration" will refer to a tuple of a particular setting of the tuning parameters. The word "output" will refer to the performance metric being optimized, such as runtime.

Table 1 summarizes the notations for GPTune which will be further explained in Section 3. As an illustrative example, the *QR factorization* routine in ScaLAPACK [3], denoted as PDGEQRF, is used as an application code to be tuned assuming a fixed core count p_{max}. Note that only independent task and tuning parameters are listed in Table 1. Other parameters, such as the number of threads *nthreads* (used in BLAS) and number of column processes p_c, can be calculated as $nthreads = \lfloor p_{max}/p \rfloor$ and $p_c = \lfloor p/p_r \rfloor$, where p is the total number of MPI processes and p_r is the number of row processes. In addition, there may be constraints on these parameters, e.g., the user can specify $p_r p_c \leq p$ (or equivalently $p_r \leq p$ using only independent parameters) to guarantee acceptable process grid dimensions. In general, each task and tuning parameter could be of type real, integer, or "categorical", i.e., a list of discrete possibilities, such as choices of algorithms.

3 GPTune Algorithms: Multitask Learning in Bayesian Optimization Framework

Bayesian optimization [1], also known as *response surface methodology*, relies on a GP surrogate model of the real objective function to optimize. The GP model is much cheaper to evaluate than the objective function, and can be iteratively updated until convergence to an optimum. The *Efficient Global Optimization* (EGO) algorithm [12] is a classical Bayesian optimization algorithm. In order to balance between exploration and exploitation (global and local search behaviors), EGO looks for a location in the search space that optimizes a certain acquisition function (e.g., *Expected Improvement* (EI)) which considers both the mean value and

	Symbol	Interpretation
	General notations	
	$\mathbb{IS}$	Task Parameter **I**nput **S**pace
	$\mathbb{PS}$	Tuning **P**arameter **S**pace
	$\mathbb{OS}$	**O**utput **S**pace
	$\mathbb{MS}$	performance **M**odel **S**pace
	α	dimension of $\mathbb{IS}$
	β	dimension of $\mathbb{PS}$
	γ	dimension of $\mathbb{OS}$
	$\tilde{\gamma}$	dimension of $\mathbb{MS}$
	δ	number of tasks
	ϵ	number of samples per task
	$T \in \mathbb{IS}^{\delta}$	array of tasks under consideration
	$X \in \mathbb{PS}^{\delta\times\epsilon}$	array of samples
	$Y \in \mathbb{OS}^{\delta\times\epsilon}$	array of output results (e.g. runtime)
	Example: ScaLAPACK PDGEQRF notations	
Task	m	number of matrix rows
	n	number of matrix columns
Tuning	b_r	row block size
	b_c	column block size
	p	number of MPI processes
	p_r	number of row processes

Table 1. Notation.

standard deviation predicted by the model. This location is then evaluated, without any gradient, through the expensive black-box objective function (i.e., run and measure the application on a parallel machine) and the corresponding value is used to update the GP model.

To describe the MLA approach in the Bayesian optimization framework, let $t \in \mathbb{IS}$ denote an input task and $x \in \mathbb{PS}$ denote a tuning parameter configuration. $\mathbb{IS}$ is the *Task Parameter Input Space* containing all the input problems that the application may encounter. (The word "input" will be dropped in the remaining document.) $\mathbb{PS}$ is the *Tuning Parameter Space* containing all the parameter configurations to be optimized, with α being the number of task parameters and β being the number of tuning parameters. We also define $\mathbb{OS}$ to be the *Output Space* of dimension γ, i.e., the number of scalar objective functions. In the following, we first describe the algorithm for single-objective autotuning ($\gamma = 1$), then describe the algorithm for multi-objective autotuning ($\gamma > 1$).

3.1 Single-objective autotuning

The MLA learning process consists of the following phases:

1. **Sampling phase.** There are two sampling steps. The first is to select a set T of δ tasks $T = [t_1; t_2; \ldots; t_\delta] \in \mathbb{IS}^{\delta}$. The goal is to get a representative sample of the variety of problems that the application may encounter, rather than focusing on a specific type of problem. Alternatively, T can represent a list of target tasks specified by the user, instead of sampling done by GPTune.
   The second sampling step is to select an initial set of tuning parameter configurations for every task. Let ϵ_{tot} denote a prescribed total number of function evaluations per task. The number of initial samples is set to $\epsilon = \epsilon_{\mathrm{tot}}/2$. For task t_i, its initial sampling X_i consists of ϵ tuning parameter configurations $X_i = [x_{i,j}]_{j\in[1,\epsilon]} \in \mathbb{PS}^{\epsilon}$. Define $X = [X_1; X_2; \ldots; X_\delta] \in \mathbb{PS}^{\delta\times\epsilon}$ to represent all the samples.
   The samples $x_{i,j}$ are evaluated through runs of the application, whose results, $y_{i,j} = y(t_i, x_{i,j}) \in \mathbb{OS}$, can be formed as $Y_i = [y_{i,j}]_{j\in[1,\epsilon]} \in \mathbb{OS}^{\epsilon}$. The set $Y = [Y_1; Y_2; \ldots; Y_\delta] \in \mathbb{OS}^{\delta\times\epsilon}$ represents the results of all these evaluations.
2. **Modeling phase.** This phase builds a Bayesian posterior probability distribution of the objective function via training a model of the black-box objective function relative to the tasks in T. We derive a single model that incorporates all the tasks, sharing the knowledge between them to better predict them all. To this end, we use the *Linear Coregionalization Model* (LCM) [13] to generalize *Gaussian Process* (GP) in the multitask setting. We first consider a single task $y(t, x) = y(x)$. A GP represents a probability model $f(x)$ for the objective function $y(x)$. It assumes that $(f(x_1), ..., f(x_\epsilon))$ is jointly Gaussian, with mean function $\mu(x)$ and covariance $\Sigma(x, x')$, which is typically a positive definite kernel function. The idea is that if x and x' are deemed similar by the kernel (i.e., with similar mean and small variance), we expect the outputs of the function at those points to be similar too. A model $f(x)$ following a GP is written as: $f(x) \sim GP(\mu(x), \Sigma)$. In general, $\mu(x)$ can be initialized to be the zero function. The modeling is done through $\Sigma(X, X)$, by maximizing the log-likelihood of the samples X with values Y on the GP. For single-task learning, the size of the covariance matrix $\Sigma(X, X)$ is $\epsilon \times \epsilon$.
   The key to LCM is the construction of an approximation of the covariance between the different outputs of the model of every $t_i \in T$. In this method, the relations between outputs are expressed as linear combinations of $Q \leq \delta$ independent *latent random functions*

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

where $a_{i,q}$ ($i \in [1, \delta]$) are hyperparameters to be learned, and u_q are the latent functions, each of which is an independent GP whose hyperparameters need to be learned as well.

Due to the independence of u_q's, the covariance between two outputs is simply the sum of auto-covariances of u_q at those two points:

$$cov(f(t_i, x), f(t_{i'}, x')) = \sum_{q=1}^{Q} (a_{i,q} a_{i',q}) cov(u_q(x), u_q(x')) \tag{2}$$

In LCM, we assume the covariance of the latent function is based on a Gaussian kernel $k_q(x, x')$:

$$cov(u_q(x), u_q(x')) = k_q(x, x') = \sigma_q^2 \exp\left(-\sum_{i=1}^{\beta} \frac{(x_i - x_i')^2}{2(l_i^q)^2}\right) \tag{3}$$

When considering all the tasks and all the samples together, the covariance matrix $\Sigma(X, X)$ is of size $\delta\epsilon \times \delta\epsilon$ with entries

$$\Sigma(x_{i,j}, x_{i',j'}) = \sum_{q=1}^{Q} (a_{i,q} a_{i',q} + b_{i,q} \delta_{i,i'}) k_q(x_{i,j}, x_{i',j'}) + d_i \delta_{i,i'} \delta_{j,j'} \tag{4}$$

where $\delta_{i,j}$ is the Kronecker delta function, and $b_{i,q}$ and d_i are diagonal regularization parameters. The learning task in the modeling phase is to find the best hyperparameters of the model, i.e. the hyperparameters σ_q, l_i^q in the Gaussian kernel Eq.(3) and coefficients $a_{i,q}, b_{i,q}, d_i$ in Eq.(4). We use a gradient-based optimization algorithm to maximize the log-likelihood of the model on the data. Specifically, we employ the limited-memory Broyden– Fletcher–Goldfarb–Shanno algorithm (L-BFGS) [19]. Note that the log-likelihood function is usually highly nonconvex, so local optimization may not converge to the/a global optimum. Therefore, one can run L-BFGS with random initial guesses for a few times, and pick the hyperparameters yielding the largest log-likelihood. Note that we can fix $\sigma_q = 1$ and d_i is only needed for the inversion of Σ.

3. **Search phase.** Once the model has been updated, the objective function values at new points $X^* = [x_1^*; x_2^*; \ldots; x_\delta^*]$ can be predicted with posterior mean $\mu^* = [\mu_1^*; \mu_2^*; \ldots; \mu_\delta^*]$ and posterior variance (confidence) $\sigma^{*2} = [\sigma_1^{*2}; \sigma_2^{*2}; \ldots; \sigma_\delta^{*2}]$ as:

$$\mu^* = \Sigma(X^*, X) \Sigma(X, X)^{-1} Y \tag{5}$$

$$\sigma^{*2} = \text{diag}\big(\Sigma(X^*, X^*) - \Sigma(X^*, X) \Sigma(X, X)^{-1} \Sigma(X^*, X)^T\big) \tag{6}$$

Here diag is the vector containing the diagonal entries. Note that the posterior variance is equal to the prior covariance minus a term that corresponds to the variance removed by observing X [9]. The mean and variance can be used to construct the EI acquisition function, which can be maximized in order to choose a new point X^* for function evaluation (additional sampling). Since the EI is cheap to compute, we can generate large numbers of samples and use global, evolutionary algorithms such as the Particle Swarm Optimization (PSO) algorithm to optimize the EI. PSO iteratively evolves a population of candidate samples towards the optimal sample with maximum EI value using a heuristic evolution strategy.

We finish one MLA iteration with one additional function evaluation at X^*, increment ϵ by 1 and move to next MLA iteration (phases 2 and 3) until ϵ reaches the prescribed sample count ϵ_{tot} (i.e., a prescribed budget of function evaluations). This iterative process is summarized as Algorithm 1.

3.2 Multi-objective autotuning

When the measured performance data consists of multiple quantities, such as (runtime, accuracy), then one can perform multi-objective autotuning. For example, in the case of runtime and accuracy, which are likely to tradeoff against one another (faster runtime leading to worse accuracy), then one may want to compute the Pareto front of these two quantities.

The MLA algorithm described in Section 3.1 can be easily extended to multi-objective, multi-task settings. Algorithm 2 describes the multi-objective extension of Algorithm 1. Let $y^s(t, x)$, $s \leq \gamma$ denote the sth objective function. Algorithm 2 essentially builds one LCM model per objective function $y^s(t, x)$ in the modeling phase. In addition, the search phase relies on multi-objective evolutionary algorithms such as non-dominated sorting generic algorithm II (NSGA-II) [5] to search for k

Algorithm 1 Bayesian optimization-based single-objective MLA

```
Sampling phase: Compute $y(t_i, x)$, $i \leq \delta$ at $\epsilon = \epsilon_{tot}/2$ initial random tuning parameter configurations for $\delta$ selected tasks.
while $\epsilon < \epsilon_{tot}$ do
    Modeling phase: Update the hyperparameters in the LCM model of $y(t_i, x)$, $i \leq \delta$ using all available data.
    Search phase: Search for an optimizer $x_i^*$ for the EI of task $t_i$, $i \leq \delta$. Let $X^* = [x_1^*; x_2^*; \ldots; x_\delta^*]$.
    Compute $y(t_i, x)$, $i \leq \delta$ at the new tuning parameter configurations $X^*$.
    $\epsilon \leftarrow \epsilon + 1$.
end while
Return the optimum tuning parameter configurations and objective function values for each task.
```

new tuning parameter configurations in each iteration. The multi-dimensional sorting is based on the Pareto dominance and crowding distance [5].

3.3 Incorporation of performance models

A performance model refers to an analytical formula for any feature (time, memory, communication volume, flop counts) of the objective function. For example, one can provide an analytical formula for the flop count when the objective function is the runtime. When available, performance models can be incorporated to build a more accurate LCM model with fewer samples needed. In what follows, the performance model incorporation is explained assuming single objective $\gamma = 1$ for simplicity.

We define $\mathbb{MS}$ to be the *performance Model Space* with dimension $\tilde{\gamma}$ being the number of models. Let $\tilde{y}(t, x)$ denote the results of the performance models for tuning parameters x and task t. Without the performance model, entries of the LCM kernel matrix represents the nonlinear inner products between points x and x' in the tuning parameter space $\mathbb{PS}$ of dimension β. One can use the values $\tilde{y}(t, x)$ as the extra features to construct an enriched space of dimension $\beta + \tilde{\gamma}$ consisting of points $[x, \tilde{y}(t, x)]$. Note that the enriched LCM matrix still has the same dimension $\epsilon\delta \times \epsilon\delta$. Once the LCM model is built, the objective function at the new point x^* can still be predicted using (5) and (6) by replacing x^* with $[x^*, \tilde{y}(t, x^*)]$.

It is worth mentioning that the performance model can have its own hyperparameters. For example, consider tuning the runtime of ScaLAPACK QR on a $m \times n$ matrix. Let $y(t, x)$ denote the objective function with task parameters $t = [m, n]$ and tuning parameters $x = [p, p_r, b_r, b_c]$. One can consider the following analytical formula as a performance model:

$$\tilde{y}(t, x) = C_{flop} \times t_{flop} + C_{msg} \times t_{msg} + C_{vol} \times t_{vol} \qquad (7)$$

with the number of floating point operations C_{flop}, the number of messages C_{msg} and the volume of messages C_{vol} given by [6]

$$C_{flop} = \frac{2n^2(3m - n)}{2p} + \frac{b_r n^2}{2p_c} + \frac{3b_r n(2m - n)}{2p_r} + \frac{{b_r}^2 n}{3p_r} \qquad (8)$$

$$C_{msg} = 3n \log p_r + \frac{2n}{b_r} \log p_c \qquad (9)$$

$$C_{vol} = \left(\frac{n^2}{p_c} + b_r n\right) \log p_r + \left(\frac{mn - n^2/2}{p_r} + \frac{b_r n}{2}\right) \log p_c \qquad (10)$$

For simplicity, we assumed $b_r = b_c$ in these formulas. The coefficients t_{flop}, t_{msg} and t_{vol} represent unknown hyperparameters in this performance model. Their estimation can be integrated into the Bayesian optimization framework in Algorithm 1 and 2 by inserting a performance model update phase before the modeling phase at line 3. The update phase can use the available ϵ samples to perform a data fitting and update the hyperparameters. The hyperparameter update is important as a bad hyperparameter estimate will result in worse tuning performance compared to no performance model.

Algorithm 2 Bayesian optimization-based multi-objective MLA

```
1: **Sampling phase:** Compute $y^s(t_i, x)$, $i \leq \delta$, $s \leq \gamma$ at $\epsilon = \epsilon_{tot}/2$ initial random tuning parameter configurations for $\delta$ selected tasks.
2: **while** $\epsilon < \epsilon_{tot}$ **do**
3:     **Modeling phase:** For each objective $s \leq \gamma$, update the hyperparameters in the LCM model of $y^s(t_i, x)$, $i \leq \delta$ using all available data.
4:     **Search phase:** Search for $k$ best tuning parameter configurations using the multi-objective EI of task $t_i$, $i \leq \delta$.
5:     Compute $y^s(t_i, x)$, $i \leq \delta$ at the $k$ new tuning parameter configurations.
6:     $\epsilon \longleftarrow \epsilon + k$.
7: **end while**
8: Return the optimum tuning parameter configurations and objective function values for each task.
```

4 Parallel implementations

GPTune supports both shared-memory and distributed-memory parallelism through dynamic thread and process management. Here we only explain the distributed-memory parallelism and refer the readers to the Users Guide [26] for more details.

4.1 Dynamic process management

By design only one MPI process executes the GPTune driver (in Python), and it dynamically creates new groups of MPI processes (workers) to speed up the objective function evaluation, modeling phase and search phase through the use of MPI spawning. To describe the spawning mechanism, we recall that there are two kinds of MPI communicators, i.e. intra- and inter-communicators. An intra-communicator consists of a group of processes and a communication context, while an inter-communicator binds a communication context with two groups (local and remote) of processes. The master process (running the GPTune driver) will call the function Spawn in mpi4py to create a group of new processes. The master process is contained in the intra-communicator "MPI_World" with only one process. The Spawn function will return an inter-communicator "SpawnedComm" that contains a local group (the master itself) and a remote group (containing the workers). The workers also have their own intra-communicator "MPI_World" and call the mpi4py function `Get_parent` that returns an inter-communicator “ParentComm" that contains a local group (the workers) and a remote group (the master). Data can be communicated between the master and workers using the inter-communicators. This scheme can be conceptually depicted in Fig. 1. In what follows, we describe the parallelism in the objective function evaluation, and modeling and search phases (in MLA) separately.

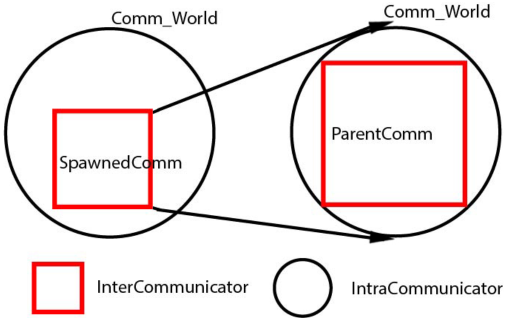

Figure 1. GPTune parallel programming model.

4.2 Launching the application code

For a distributed-memory application code, GPTune will call the objective function that spawns the application code with given task and tuning parameters. The MPI count can be passed to the application code as an argument of the `Spawn` function, and the thread count can be passed using the environment variable. Depending on how the application code is implemented, one can pass the parameters using command line, environment variables, or input files stored on the disk. To collect the returning value(s) from the workers, one can choose to read from the individual log file per function execution or communicate using the inter-communicators. GPTune also supports calling multiple function evaluations concurrently, we refer the reader to the Users Guide [26] for details.

4.3 Modeling and search phases of MLA

The modeling phase, described in Section 3.1, uses the L-BFGS algorithm to find a set of LCM hyperparameters that maximizes the log-likelihood function using selected objective function samples. The GPTune implementation can choose n_{start} random starting guesses of the hyperparameters, each used by L-BFGS to search for the maximum log-likelihood. GPTune then chooses the set of hyperparameters that yields the best log-likelihood.

The current implementation supports two levels of parallelism in this phase: (1) The number of n_{start} random starts for the hyperparameters and corresponding L-BFGS optimization are distributed over a prescribed number of MPI processes. (2) For each L-BFGS optimization, the factorization of the covariance matrix is parallelized over a prescribed number of MPI processes. GPTune uses MPI spawning to support the distributed-memory parallelism over the random starts and factorization of the covariance matrix.

The search phase uses evolutionary algorithms to search for the next sample point for each task (see Section 3.1 for details). The multi-task search can be parallelized over the δ tasks using a user-specified number of MPI processes. GPTune uses MPI spawning to support the distributed-memory parallelism over the tasks.

5 Related Work

The fundamental aspect of autotuning is optimization, i.e., finding a parameter configuration so that it solves a given input problem optimally. The nature of autotuning makes this optimization problem lie within the family of black-box optimization problems, which are among the hardest to solve.

The simplest black-box optimization methods often tried first are: (1) The *exhaustive search* (and its variant *grid search*), which tries all (or subset of all, respectively) possible combinations of all possible values of the parameters and selects the best performing one. These quickly become intractable when the number of parameters increases, due to curse of dimensionality [2]; (2) The stochastic *random search*, which randomly selects the value of each parameter in order to generate candidate solutions, then selects the best performing candidate.

The more advanced optimization methods can usually be categorized as model-free (non-Bayesian) optimization or Bayesian optimization. There are two main families of model-free optimization approaches. The *global* approaches explore the entire search space and attempt to find a balance between exploration of new regions of the search space and exploitation of the data gathered to give more attention to the promising regions of space. Examples of such methods are *Simulated Annealing* [25], *Genetic Algorithms* [27] and PSO [15]. In contrast, the *local* approaches attempt to improve upon previous solutions by exploring their neighboring region only until converging to a local minimum. Examples of such approaches are *Nelder–Mead simplex* [11] and *Orthogonal Search* [4].

OpenTuner [10] is one of the state-of-the-art general purpose autotuners, which implements most of the above model-free optimization algorithms. Moreover, at a higher level, it relies on meta-heuristics to solve a multi-armed bandit problem [14] where application runtime (function evaluation) is the resource to be allocated. Specifically, it allocates and distributes the function evaluations over a collection of the aforementioned optimization methods in multiple "arms" in order to adaptively select the best performing method, which is used to solve the autotuning optimization problem.

Several other non-Bayesian black-box optimization packages for autotuning exist in the literature. Particularly, SuRf [23] uses random forests to model the performance of an application and find its optimum. One of its main strengths is its ability to handle categorical parameters (choices) in an elegant way.

HpBandSter [8] is another state-of-the-art general purpose autotuner. It combines Bayesian optimization and bandit-based methods. The earlier hyperband [18] is a multi-armed bandit strategy that dynamically allocates resources to a set of random configurations and uses successive halving to stop poorly performing configurations. HpBandSter infuses a model-based search (Bayesian optimization) algorithm instead of random selection of configurations at the beginning of each hyperband iteration. In the Bayesian optimization aspect, HpBandSter differs from GPTune in that it uses a kernel density estimator, called Tree Parzen Estimator (TPE) to select a new configuration to evaluate, instead of directly optimizing EI as GPTune does. This is faster, but less accurate.

Recent autotuning work using a Bayesian optimization framework is presented by Menon et al. [21]. Similar to HpBandSter, it also uses the TPE method to search for the next configuration to evaluate. Unlike HpBandSter, it does not use the multi-armed bandit framework.

In Section 6.6, we compare GPTune only with OpenTuner and HpBandSter, because they are publicly available, and we utilize the MPI spawning approach to call distributed-memory MPI codes.

6 Experimental results

This section presents the experimental results of the methods described in this paper together with the comparisons with state-of-the-art autotuning tools.

6.1 GPTune software

GPTune software is freely available at https://github.com/gptune/GPTune. The software is implemented in Python (mostly) and C, and depends on the following Python packages: numpy, scikit-learn, scipy, GPy, lhsmdu, pygmo, scikit-optimize, mpi4py and autotune. In addition, the C implementation requires OpenMPI/4.0, BLAS/LAPACK, and ScaLAPACK. All results in this section can be reproduced following the instructions and example scripts at the Github repository.

To make it easier for users to try different autotuners, our interface allows the user to invoke them as well. So far, OpenTuner [10], HpBandSter [8], and ytopt [31] are supported.

6.2 Parallel machine and HPC codes

We use the Cori machine at NERSC,[1] which is a Cray XC40 system with 2,388 Haswell nodes, each with two 16-core Intel Xeon E5-2698v3 processors and 128GB of 2133MHz DDR4 memory.

Table 2 lists the parallel application codes used in our experiments. Unless otherwise stated, the objective function/performance metric is the runtime of the application. The routine PDGEQRF from the ScaLAPACK library [3] performs dense QR factorization of a matrix of size $m \times n$. The ScaLAPACK routine PDSYEVX computes selected eigenvalues and, optionally, eigenvectors of a real symmetric matrix. SuperLU_DIST is a sparse direct solver for nonsymmetric linear systems [16, 29, 30]. In ScaLAPACK, a dense matrix is partitioned into blocks. The processes are arranged in a 2D process grid. The matrix blocks are distributed in the 2D process grid in a block-cyclic fashion in both dimensions. SuperLU_DIST uses a similar 2D block-cyclic distribution for the L and U factored matrices, except that the block partition follows supernode partition with non-uniform block sizes, depending on matrix sparsity pattern. Nevertheless, for both dense and sparse codes, the shape of the 2D process grid $p = p_r \times p_c$ is a tuning parameter. More specially, $t = [m, n]$, $x = [b_r, b_c, p, p_r]$ for PDGEQRF and PDSYEVX. For PDSYEVX, we enforce $m = n$ and $b_r = b_c$. We use the matrix name to define a task for SuperLU_DIST, and the tuning parameters are $x =$ [COLPERM, LOOK, p, p_r, NSUP, NREL] where COLPERM, LOOK, NSUP and NREL represent the type of column permutation to preserve sparsity of the LU factors, number of look-ahead columns in the pipeline factorization, the maximum supernode size and the relaxed supernode size corresponding to the bottom subtrees of the elimination tree. To cope with runtime noise in the measurements, all the runs of PDGEQRF and PDSYEVX were performed 3 times, and the minimal runtime was selected.

The package hypre [7] contains several families of parallel algebraic multigrid preconditioners and solvers for large-scale sparse linear systems. Here we focus on autotuning the performance of GMRES with the BoomerAMG preconditioner for solving the Poisson equation on structured 3D grids. We define a task as $t = [n_1, n_2, n_3]$ where n_i denotes the grid size in ith dimension. The process is arranged in a 3D process grid $p = p_1 \times p_2 \times p_3$ where p_i denotes the number of processes in ith dimension. In addition to the process grid, we consider a total of 12 tuning parameters of integer and real types, including choice of coarsening algorithms, smoothers and

[1] http://www.nersc.gov/users/computational-systems/cori/

interpolation operators, and their corresponding parameters.

Both M3D_C1 [20] and NIMROD [22] solve the extended magnetohydrodynamic equations. They are primarily used for calculating the equilibrium, stability, and dynamics of fusion plasmas, and are critical simulation codes for designing the reactor-scale tokamaks such as ITER. The tokamak device has 3D torus geometry. M3D_C1 uses C^1 finite element discretization in three dimensions. NIMROD uses spectral finite element discretization in two dimensions and finite Fourier series in the third dimension. Both are time-marching codes and solve nonsymmetric sparse linear systems with preconditioned GMRES, for which multiple instances of SuperLU_DIST are used to solve the poloidal plane problems as a block Jacobi preconditioner. We fix the geometry model, its discretizations and MPI count p, and define a task t as the number of time steps. For M3D_C1, the tuning parameters are x = [ROWPERM, COLPERM, p_r, NSUP, NREL] where ROWPERM is the type of row permutation to maintain numerical stability; for NIMROD, the tuning parameters are x = [ROWPERM, COLPERM, p_r, NSUP, NREL, nxbl, nybl] where nxbl and nybl are block sizes for assembling the NIMROD matrices. Here ROWPERM and COLPERM are of categorical type.

Application	Description	β
analytical	sequential function	1
ScaLAPACK PDGEQRF	dense QR factorization	3
ScaLAPACK PDSYEVX	dense symmetric eigenvalue	3
SuperLU_DIST	sparse direct linear solver	6
hypre	algebraic multigrid solver	12
M3D_C1	fusion plasma	5
NIMROD	fusion plasma	7

Table 2. Descriptions of the HPC codes used in the tuning experiments. All but the first one are MPI codes. β is the number of parameters to be tuned.

6.3 Parallel speedups of GPTune

Consider the following analytical problem to be tuned, with the objective function given explicitly as

$$y(t,x) = 1 + e^{-(x+1)^{t+1}} \cos(2\pi x) \sum_{i=1}^{3} \sin\left(2\pi x (t+2)^i\right) \quad (11)$$

where t and x (both of real type) denote the task and tuning parameters. Note that this function is highly nonconvex, representing a very hard problem for black-box optimization. We are interested in finding the (global) minimum for $x \in [0,1]$ for multiple tasks t. Fig. 2 plots $y(t,x)$ for four values of t and marks the function minimum.

We evaluate the parallel performance of the MLA algorithm on 1 Cori node using $\delta = 20$ tasks. In Fig. 3, we plot the runtime of the modeling and search phases using 1 and 32 MPIs. For simplicity, we set the initial random sample count to $\epsilon = \epsilon_{tot} - 1$ (i.e., only one MLA iteration is performed). Note that with 1 MPI process, the modeling and search phases obey the theoretical scaling of $O(\epsilon_{tot}^3 \delta^3)$ and $O(\epsilon_{tot}^2 \delta^2)$, respectively. As we increase the number of total samples ϵ_{tot} from 20 to 320 (with the LCM kernel matrix size changing from 400 to 6400), 32X (comparing the two blue curves) and 11X (comparing the two black curves) speedups are observed for the modeling and search phases, respectively. For the modeling phase, we parallelized the factorization of the covariance matrix using ScaLAPACK, so an ideal speedup is achieved for large enough covariance matrices; for the search phase, we distributed the δ tasks over MPI ranks, so the speedup is at most $\delta = 20$.

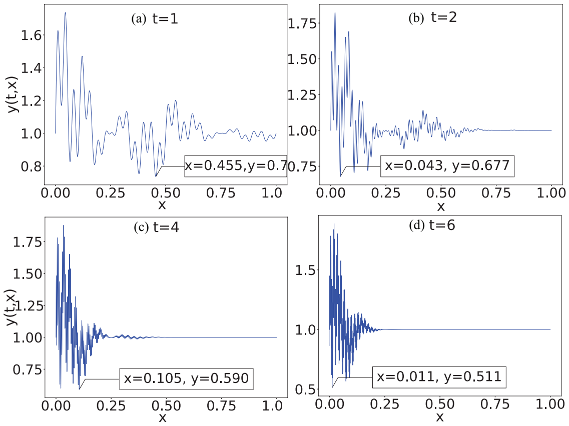

Figure 2. The objective functions in (11) for four task parameter values t.

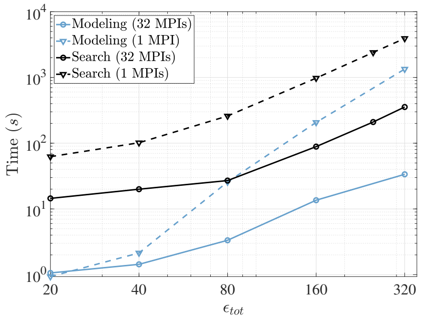

Figure 3. Modeling and search time for 1 and 32 MPI processes using the analytical objective function in Eq 11.

6.4 Advantage of using performance models

Next, we evaluate the effects of the performance models using the analytical function in Eq. (11) and the ScaLAPACK PDGEQRF example. We expect that the coarse performance model will be useful when the objective function is highly non-convex with only a small number of samples available.

For the analytical function, we consider a performance model $\tilde{y}(t,x) = (1+0.1\times r(x))y(t,x)$ (the model is the objective with random scaling factors). Here, $r(x)$ is a random number drawn from the normal distribution $\mathcal{N}(0,1)$. We generate $\delta = 20$ tasks $t = 0, 0.5, \ldots, 9.5$, and compare the performance of MLA with and without the performance model. The ratios of objective minimum found by MLA with $\epsilon_{tot} = 20, 40, 80$ are plotted in Fig. 4 (left). For all tasks, the performance model yields an equal or better minimum (ratio ≥ 1). Also, the ratios of the minimum found by MLA and the true minimum are also plotted. Clearly the true minimum is attained for most data points. Note that the tasks are sorted by t in an ascending order. From (11) and Fig. 2, larger t yields a more complex-shaped objective function. It is clear that the noisy performance model is more beneficial for larger t. Also, the ratios are in general higher for smaller ϵ_{tot}. In other words, with the noisy performance model, GPTune requires significantly fewer samples to build an accurate LCM model.

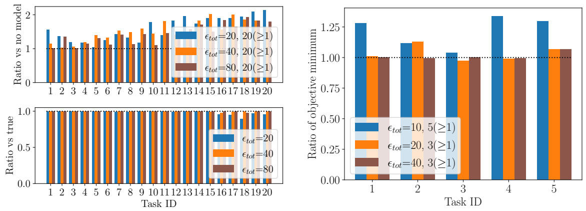

Figure 4. Left: analytical function (Eq. 11), ratios of the tuned minimum without performance model over that with performance model (upper), or ratios of the true minimum over the tuned minimum with performance model (lower). Right: ScaLAPACK PDGEQRF, ratios of the tuned minimum without the performance model over that with the performance model. $x(>= 1)$ in the legends means that there are x tasks out of the δ tasks with the ratio greater than 1.

For the PDGEQRF example, we consider the performance model in Eq. (7) with on-the-fly hyperparameter estimation. We generate 5 random tasks with $m, n < 20000$ and run MLA without and with the performance model using 16 Cori nodes. The ratios of the best runtime are plotted in Fig. 4 (right). When $\epsilon_{tot} = 10$, the coarse performance model yields an up to 35% improvement. As ϵ_{tot} becomes larger (20 and 40), MLA has sufficient samples to build the accurate LCM model, and the performance model has less effect.

6.5 Efficiency of multi-task learning

Next, we compare the performance of the GPTune MLA algorithms with single-task (δ=1) and multitask (δ>1) settings. We enforce that the total number of function evaluations $\delta \times \epsilon_{tot}$ is the same between the single-task and multitask settings.

	PDGEQRF			
	total	objective	modeling	search
Single-task	8831.6	8800.1	6.7	24.6
Multitask	7315.1	5699.9	1529.4	85.6
	PDSYEVX			
	total	objective	modeling	search
Single-task	9831.1	9032.1	632.1	166.7
Multitask	6595.1	6334.8	74.7	185.5

	M3D_C1		NIMROD	
	minimum	total	minimum	total
Single-task	11.19	12310	112.7	14710
Multitask	11.17	7797	112.8	9559

Table 3. Upper: Runtime of different phases for PDGEQRF and PDSYEVX. Lower: Best runtime and the total time spent in the application code for M3D_C1 (t = 3) and NIMROD (t = 15).

For the PDGEQRF example, we use 64 Cori nodes (2048 cores) assuming a fixed budget of $\delta \times \epsilon_{tot} = 100$. For δ=1, we consider the task ($m = 23324, n = 26545$); for δ=10, we also consider 9 other tasks that are randomly generated with $m, n < 40000$. Table 3 shows the runtime breakdown of the single-task and multitask tuning algorithms. The multitask algorithm requires less objective evaluation time as it involves 9 other less expensive tasks. Fig. 5(left) plots the best and worst runtime for all 10 tasks identified by their flop counts. The multitask MLA not only achieves a very similar minimum to the single-task MLA (for ($m = 23324, n = 26545$)), but also finds minima for all the other 9 tasks. Note that PDGEQRF can achieve 3.6 TFLOPS with optimal tuning parameters.

For the PDSYEVX example, we use 1 Cori node. For δ=1, we consider the task ($m = 7000$) with $\epsilon_{tot} = 90, 180$. For $\epsilon_{tot} = 90$, the best runtime from $\epsilon_{tot}/2$ initial samples and all ϵ_{tot} samples is 9.63 and 9.41. For $\epsilon_{tot} = 180$, the best runtime from $\epsilon_{tot}/2$ initial samples and all ϵ_{tot} samples is 9.47 and 9.36. This clearly demonstrates the usefulness of Bayesian optimization. For δ=9, we also add other 8 tasks with $3000 \leq m \leq 7000$ with $\epsilon_{tot} = 10, 20$. Fig. 5 (right) plots the best and worst runtime among all ϵ_{tot} samples for each task. The best runtime scales as $O(m^3)$; the single-task and multitask settings attain similar best runtime for $m = 7000$; using larger ϵ_{tot} slightly improves the best runtime. Table 3 shows the runtime breakdown of the single-task and multitask tuning algorithms.

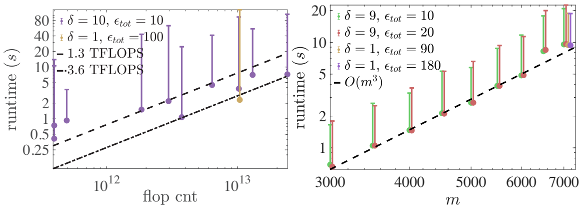

Figure 5. Left: best and worst runtime of PDGEQRF for $\delta = 10$ tasks with $\epsilon_{tot} = 10$ using 64 Cori nodes (2048 cores). The tasks are sorted by their flop counts. Right: best and worst runtime of PDSYEVX for $\delta = 9$ tasks with $\epsilon_{tot} = 10, 20$ using 1 Cori node.

For the M3D_C1 and NIMROD examples, we use the number of time steps as the task parameter t. Specifically, a practical M3D_C1 or NIMROD simulation can require hundreds of time steps to compute meaningful physical quantities, and it can be prohibitively expensive to directly use it for autotuning. Therefore, using MLA one can run applications with both small and large number of steps to reduce the tuning time. For M3D_C1, we compare single-task ($t = 3$ and $\epsilon_{tot} = 80$) with multitask ($t = 1, 1, 1, 3$ and $\epsilon_{tot} = 20$) settings. Each simulation requires 1 Cori node. For NIMROD, we compare single-task ($t = 15$ and $\epsilon_{tot} = 80$) with multitask ($t = 3, 3, 3, 15$ and $\epsilon_{tot} = 20$) settings. Each simulation requires 6 Cori nodes. As Table 3 shows, multitask tuning obtained minimum runtimes similar to the single-task tuning but significantly reduced the total function evaluation time. Our tuning shows a 15% to 20% improvement compared to default. Once the optimal tuning parameters are discovered, they can be used in the realistic simulation requiring hundreds of time steps.

6.6 Performance comparison with other tuners

Next, we compare the performance of GPTune MLA with two other autotuners, OpenTuner and HpBandSter. OpenTuner dynamically selects the optimization techniques if none is specified by users. For HpBandSter, we disabled the multi-armed bandit feature since it requires running applications with varying fidelity/budgets. More details regarding the configuration of OpenTuner and HpBandSter can be found in the GPTune software repository.

For the PDGEQRF example, we generate $\delta = 10$ random tasks with $m, n < 20000$ and run the three tuners with $\epsilon_{tot} = 10$ using 64 Cori nodes (2048 cores). Since OpenTuner and HpBandSter do not support multitask learning, we run them separately on each task. The ratios of the best runtimes of the two other tuners over those of GPTune are plotted in Fig. 6. GPTune outperforms OpenTuner up to 4.9X for 7 tasks and HpBandSter up to 2.9X for 8 tasks.

For the SuperLU_DIST example, we consider $\delta = 7$ matrices from the PARSEC group of SuiteSparse Matrix Collection [28], and tune the factorization time with $\epsilon_{tot} = 20$ using 32 Cori nodes (1024 cores). The ratios of the best runtimes of the two other tuners over those of GPTune are plotted in Fig. 6. GPTune outperforms OpenTuner up to 1.6X for 6 tasks and HpBandSter up to 1.3X for 7 tasks.

For the hypre example, we generate $\delta = 30$ random tasks with $10 \leq n_1, n_2, n_3 \leq 100$ and run the three tuners with $\epsilon_{tot} = 10, 20, 30$ using 1 Cori node and 4 Cori nodes respectively. Table 4 compares both final performance and anytime performance with OpenTuner and HpBandSter. The final performance is measured by the metric WinTask, which is the percentage of tasks that GPTune finds a better objective minimum and outperforms the other tuner. In addition to the final performance, the anytime performance of the three tuners (i.e., quality of the function minimum when the tuning is terminated at any time) is also compared. For each tuner, we define the stability for each task t_i as $\text{mean}(y^*(t_i, x_1), \ldots, y^*(t_i, x_{\epsilon_{tot}}))/y^*(t_i)$, here $y^*(t_i, x_j)$ is the best runtime among samples 1 to j, and $y^*(t_i)$ is the best runtime over ϵ_{tot} samples of all tuners. In an MLA experiment, the anytime performance of a tuner is then measured by the average stability among all tasks. It is shown in Table 4 that GPTune outperforms other tuners on both metrics on all experiments.

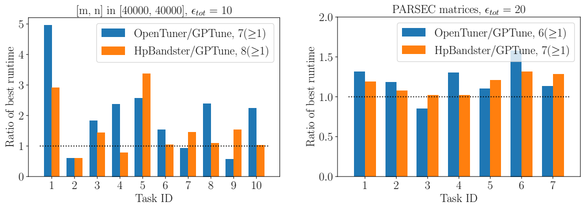

Figure 6. Ratios of the objective minimum between GPTune and the other tuners. Left: PDGEQRF with $\delta = 10$ tasks and $\epsilon_{tot} = 10$ using 64 Cori nodes (2048 cores). Right: SuperLU_DIST with $\delta = 7$ matrices and $\epsilon_{tot} = 20$ using 32 Cori nodes (1024 cores). The matrices are Si2, SiH4, SiNa, Na5, benzene, Si10H16 and Si5H12. The task count out of the δ tasks with ratio ≥ 1 is shown in the legends.

6.7 Capability of multi-objective tuning

Finally, we illustrate the multi-objective feature of GPTune for tuning the factorization performance in package SuperLU_DIST [17]. We consider $\gamma = 2$ objectives

Setups		WinTask		mean(stability)		
nodes	ϵ_{tot}	vs OT	vs HB	GPTune	OT	HB
1	10	74%	73%	**1.27**	1.97	1.58
1	20	63%	70%	**1.21**	1.46	1.38
1	30	60%	80%	**1.23**	1.56	1.37
4	10	60%	67%	**1.38**	1.79	1.77
4	20	66%	83%	**1.29**	1.90	1.64
4	30	64%	63%	**1.33**	1.51	1.33

Table 4. Comparisons of the final performance and anytime performance between GPTune and other tuners (OT: OpenTuner, HB: HpBandSter) on several `hypre` MLA experiments. WinTask measures GPTune's final performance compared with another tuner, the higher the better, while mean(stability) measures the anytime performance of a tuner, the smaller the better.

(time, memory) representing factorization time and memory. We consider both the single-task and multitask tuning tests for matrices from the PARSEC group using 8 Cori nodes.

For single task, we consider the matrix "Si2" and compare the performance of single-objective (i.e., time or memory) and multi-objective tuning. For example, single-objective memory tuning means minimizing the memory usage ignoring the impact on runtime. Table 5 lists the default and optimal tuning parameters; the optimal ones are vastly different from the default ones. Fig. 7 plots the objective function values using the default tuning parameters, and those optimal ones by the GPTune single-objective and multi-objective MLA algorithms with ϵ_{tot} = 80. The multi-objective MLA algorithm returns multiple tuning parameter configurations and their objective function values (in black), among which no data point Pareto-dominates over any other. In other words, the black dots lie on the Pareto front. We see that the single-objective minima (in yellow and magenta) lie on or near the Pareto front formed by the multi-objective minima (in black). Not surprisingly, the default objective values (in cyan) are far from optimal in either dimension. The tuned performance can achieve a 83% improvement in time or 93% improvement in memory compared to default.

Next, we consider 8 matrices and compare the performance of single-task (calling GPTune with δ = 1 for each matrix) and multitask (calling GPTune with δ = 8) tuning with ϵ_{tot} = 80. Fig. 7 plots the Pareto fronts using both tuners. As expected, there are very few data points returned by the single-task tuner that Pareto-dominates over those returned by the multitask tuner.

	COLPERM	LOOK	p	p_r	NSUP	NREL
Default	4	10	256	16	128	20
Time	2	6	216	149	295	37
Memory	2	5	193	20	31	22

Table 5. Default tuning parameters and optimal ones returned by the GPTune single-objective MLA algorithm.

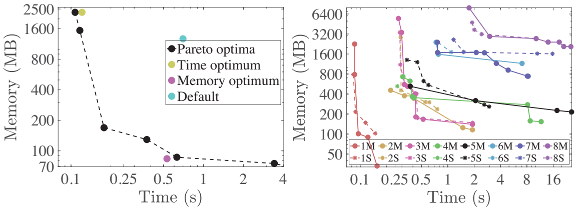

Figure 7. Logarithmic plots of the optimal objective functions values (factorization time and memory of `SuperLU_DIST` with 8 Cori nodes) discovered by GPTune. Left: matrix Si2. Right: 8 PARSEC matrices: Si2, SiH4, SiNa, Na5, benzene, Si10H16, Si5H12, SiO. "M" and "S" denote multitask and single-task, respectively.

7 Conclusions

GPTune is a multitask learning and Bayesian optimization-based autotuner well-suited for tuning exascale application codes ranging from high-performance mathematical libraries to production-level scientific simulation codes. To the best of our knowledge, GPTune is the first distributed-memory parallel autotuner. In addition, GPTune supports advanced features such as multi-objective autotuning and incorporation of coarse performance models. When compared to other state-of-the-art autotuners such as OpenTuner and HpBandSter, GPTune achieves significantly better final and anytime performance, particularly with a small budget.

Acknowledgments

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] B. Shahriari and K. Swersky and Z. Wang and R. P. Adams and N. de Freitas. 2016. Taking the Human Out of the Loop: A Review of Bayesian Optimization. *Proceedings of the IEEE* 104, 1 (Jan 2016), 148–175.

[2] Richard Bellman. 1957. *Dynamic Programming* (1 ed.). Princeton University Press, Princeton, NJ, USA. http://books.google.com/books?id=fyVtp3EMxasC&pg=PR5&dq=dynamic+programming+richard+e+bellman&

client=firefox-a#v=onepage&q=dynamic%20programming%20richard%20e%20bellman&f=false

[3] Blackford, L. S. and Choi, J. and Cleary, A. and D'Azevedo, E. and Demmel, J. and Dhillon, I. and Dongarra, J. and Hammarling, S. and Henry, G. and Petitet, A. and Stanley, K. and Walker, D. and Whaley, R. C. 1997. *ScaLAPACK Users' Guide*. Society for Industrial and Applied Mathematics, Philadelphia, PA.

[4] Chan, Timothy M. and Larsen, Kasper Green and Pătraşcu, Mihai. 2011. Orthogonal Range Searching on the RAM, Revisited. In *Proceedings of the Twenty-seventh Annual Symposium on Computational Geometry* (Paris, France) *(SoCG '11)*. ACM, New York, NY, USA, 1–10.

[5] Kalyanmoy Deb, Amrit Pratap, Sameer Agarwal, and TAMT Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE transactions on evolutionary computation* 6, 2 (2002), 182–197.

[6] James Demmel, Laura Grigori, Mark Hoemmen, and Julien Langou. 2012. Communication-optimal parallel and sequential QR and LU factorizations. *SIAM Journal on Scientific Computing* 34, 1 (2012), A206–A239.

[7] Robert D. Falgout and Ulrike Meier Yang. 2002. hypre: A Library of High Performance Preconditioners. In *Computational Science — ICCS 2002*, Peter M. A. Sloot, Alfons G. Hoekstra, C. J. Kenneth Tan, and Jack J. Dongarra (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 632–641.

[8] Stefan Falkner, Aaron Klein, and Frank Hutter. 2018. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In *Proceedings of the 35th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 80)*, Jennifer Dy and Andreas Krause (Eds.). PMLR, Stockholmsmässan, Stockholm Sweden, 1437–1446. http://proceedings.mlr.press/v80/falkner18a.html

[9] P.I. Frazier. 2018. A Tutorial on Bayesian Optimization. https://arxiv.org/abs/1807.02811.

[10] Jason Ansel and Shoaib Kamil and Kalyan Veeramachaneni and Jonathan Ragan-Kelley and Jeffrey Bosboom and Una-May O'Reilly and Saman Amarasinghe. 2014. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*. [Association for Computing Machinery], Edmonton, Canada, 303–316. http://groups.csail.mit.edu/commit/papers/2014/ansel-pact14-opentuner.pdf

[11] John A. Nelder and Roger Mead. 1965. A simplex method for function minimization. *Computer Journal* 7 (1965), 308–313.

[12] Donald R. Jones, Matthias Schonlau, and William J. Welch. 1998. Efficient Global Optimization of Expensive Black-Box Functions. *Journal of Global Optimization* 13, 4 (01 Dec 1998), 455–492. https://doi.org/10.1023/A:1008306431147

[13] Andre G Journel and Charles J Huijbregts. 1978. *Mining geostatistics*. Vol. 600. Academic press London, London.

[14] Katehakis, Michael N. and Veinott,Jr., Arthur F. 1987. The Multi-armed Bandit Problem: Decomposition and Computation. *Mathematics of Operations Research* 12, 2 (May 1987), 262–268. https://doi.org/10.1287/moor.12.2.262

[15] J. Kennedy and R. Eberhart. 1995. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, Vol. 4. IEEE, Perth, 1942–1948 vol.4.

[16] X. S. Li and J. W. Demmel. 2003. SuperLU_DIST: A Scalable Distributed-Memory Sparse Direct Solver for Unsymmetric Linear Systems. *ACM Trans. Math. Software* 29, 2 (June 2003), 110–140.

[17] Xiaoye S. Li and James W. Demmel. 2003. SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.* 29, 2 (June 2003), 110–140. http://doi.acm.org/10.1145/779359.779361

[18] Li, Lisha and Jamieson, Kevin and DeSalvo, Giulia and Rostamizadeh, Afshin and Talwalkar, Ameet. 2017. Hyperband: A Novel Bandit-based Approach to Hyperparameter Optimization. *J. Mach. Learn. Res.* 18, 1 (Jan. 2017), 6765–6816. http://dl.acm.org/citation.cfm?id=3122009.3242042

[19] Dong C. Liu and Jorge Nocedal. 1989. On the Limited Memory BFGS Method for Large Scale Optimization. *MATHEMATICAL PROGRAMMING* 45 (1989), 503–528.

[20] m3dc1. 2004. M3D-C1. https://w3.pppl.gov/~nferraro/m3dc1.html

[21] H. Menon, A. Bhatele, and T. Gamblin. 2020. Auto-tuning Parameter Choices in HPC Applications using Bayesian Optimization. In *34th IEEE International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE, Virtual, 831–840.

[22] nimrod. 2004. NIMROD. https://nimrodteam.org/

[23] P. Balaprakash. 2015. SuRF: Search using Random Forest. https://github.com/brnorris03/Orio/tree/master/orio/main/tuner/search/mlsearch.

[24] S. J. Pan and Q. Yang. 2010. A Survey on Transfer Learning. *IEEE Transactions on Knowledge and Data Engineering* 22, 10 (Oct 2010), 1345–1359.

[25] S. Kirkpatrick and C. D. Gelatt and M. P. Vecchi. 1983. Optimization by simulated annealing. *SCIENCE* 220, 4598 (1983), 671–680.

[26] Wissam M. Sid-Lakhdar, James W. Demmel, Xiaoye S. Li, Yang Liu, and Osni Marques. 2020. GPTune Users Guide. https://github.com/gptune/GPTune/tree/master/Doc.

[27] Srinivas, M. and Patnaik, Lalit M. 1994. Genetic Algorithms: A Survey. *Computer* 27, 6 (June 1994), 17–26.

[28] suitsparse. 2018. SuiteSparse Matrix Collection. https://sparse.tamu.edu/.

[29] Xiaoye S. Li. 2005. An overview of SuperLU: Algorithms, implementation, and user interface. *ACM Trans. Math. Softw.* 31, 3 (2005), 302–325. https://doi.org/10.1145/1089014.1089017

[30] Xiaoye Sherry Li and James Demmel and John R. Gilbert and Laura Grigori and Meiyue Shao. 2011. SuperLU. In *Encyclopedia of Parallel Computing*. Springer, Boston, 1955–1962. https://doi.org/10.1007/978-0-387-09766-4_95

[31] ytopt. 2019. ytopt: Machine-learning-based search methods for autotuning. https://github.com/ytopt-team/ytopt.

[32] Yu Zhang and Qiang Yang. 2017. A Survey on Multi-Task Learning. *CoRR* abs/1707.08114 (2017), 1–1. arXiv:1707.08114 http://arxiv.org/abs/1707.08114

A Artifact Setup and Evaluation

A.1 Abstract

This artifact describes how to test the basic functionality of GPTune, generate figures in the paper, and perform the numerical experiments with smaller core counts, sample counts, and cheaper applications than those in the paper, using a personal computer. The expensive experiments can be reproduced if the user has access to the NERSC Cori supercomputer.

A.2 Description

- Software prerequisites: APT (Ubuntu-like), homebrew (MacOS) or Docker. The rest are installed by the provided scripts.
- Hardware: personal computer with at least 4 cores, or Cori Haswell with 64 nodes
- Metrics: Objective minimization (e.g., application runtime)
- How much time is needed to prepare workflow (approximately)?: Docker image: none. Manual installation: 1 hour.
- How much time is needed to complete experiments (approximately)?: 2 hours on a personal computer
- Publicly available?: Yes.
- Code licenses (if publicly available)?: BSD

A.3 Installation

A.3.1 Docker image

1. docker pull liuyangzhuan/gptune:1.2
2. docker run -ti liuyangzhuan/gptune:1.2

A.3.2 Manual installation

1. git clone https://github.com/gptune/GPTune.git
2. cd GPTune
3. git checkout 1b897b05017
4. run one of the following installation scripts:
    - Ubuntu-like: sudo sh config_ubuntu_moreinstall.sh
    - MacOS: zsh config_macbook_moreinstall.zsh
    - Cori Haswell: bash config_cori_haswell_gnu.sh

A.4 Testing the setup

The functionality of GPTune can be tested using one of the following scripts.

- Docker: bash run_ubuntu_moreinstall.sh
- Ubuntu-like OS: bash run_ubuntu_moreinstall.sh
- MacOS: zsh run_macbook_moreinstall.zsh
- Cori Haswell: allocate one Cori node and bash run_cori_moreinstall.sh

Three examples are illustrated: 1. Minimizing the analytical function with one task $t = 0$ in (11). 2. Tuning runtime of PDGEQRF on a small matrix ($m, n < 1300$). 3. Tuning runtime of SuperLU_DIST on a small matrix "big.rua". The command line options in the run_*.sh can be changed if of interest to the users. The optimal tuning parameters and objective function values are printed after "Popt" and "Oopt" for each task. The tuner time breakdown is printed after "stats:". For more details about interpreting the runlog, see the userguide [26], Section 5.1.1 Listing 4 for an example of tuning PDGEQRF.

A.5 Reproducing the experiments and figures

This subsection provides scripts to generate the paper figures, and rerun most experiments on a much smaller scale. The scripts are the same as those in Section A.4. Take run_ubuntu_moreinstall.sh for Ubuntu-like OS as an example, change "test" at line 27 to one of the following: "Fig.2", "Fig.3", "Fig.3_exp", "Fig.4", "Fig.4_exp", "Fig.5", "Fig.5_exp", "Fig.6", "Fig.6_exp", "Fig.7", "Fig.7_exp", "Tab.4_exp". See the descriptions in this script for their usage. Essentially, "Fig.*" generates the corresponding figures in the paper, and "*_exp" performs the corresponding experiments on a much smaller scale.