GPTune Users Guide*

Wissam M. Sid-Lakhdar, † James W. Demmel‡ Xiaoye S. Li† Yang Liu† Osni Marques†

June 12, 2020

Contents

*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.

†Lawrence Berkeley National Laboratory, MS 50A-3111, 1 Cyclotron Rd, Berkeley, CA 94720. (wissam.sidlakhdar@gmail.com, xsli@lbl.gov, liuyangzhuan@lbl.gov, oamarques@lbl.gov)

‡Computer Science Division, University of California, Berkeley, CA 94720. (demmel@cs.berkeley.edu).

Abstract

GPTune is an autotuning framework that relies on multitask and transfer learning to help solve an underlying black-box optimization problem. GPTune is part of the xSDK4ECP project supported by the Exascale Computing Project (ECP).

Multitask learning and transfer learning have proven to be useful in the field of machine learning when additional knowledge is available to help a prediction task. We aim at deriving methods following these paradigms for use in autotuning, where the goal is to find the best parameters for optimal performance of an application treated as a black-box function.

We assume that evaluations are expensive, and so try to minimize the number of evaluations.

1 Introduction

The goal of autotuning is to automatically choose tuning parameters to optimize the performance of an application. Performance is most often measured by runtime, but any quantitative, measurable quantity is possible, such as the number of messages communicated, amount of memory used, accuracy, etc. Autotuning is particularly challenging when the number of (combinations of) tuning parameter values is large, the performance is a complicated and hard-to-model function of all the tuning parameters, and running the application in order to measure the actual performance is expensive.

GPTune addresses these autotuning challenges by using Bayesian optimization (Gaussian Processes) to build a performance model (by running the application and measuring its performance at a few carefully chosen tuning parameter values), and optimizing the model (eg choosing the tuning parameters to minimize the runtime predicted by the model). In the simplest case, the user has a single task t (eg a linear algebra operation on a matrix of a certain size), the model $f(x)$ predicts the true runtime $y(x)$ as a function of a tuning parameter x (eg a block size), and GPTune chooses x to minimize $f(x)$.

GPTune goes beyond this simple case by providing both Multitask Learning Autotuning (MLA) and Transfer Learning Autotuning (TLA). MLA means using performance data from multiple tasks (eg a fixed linear algebra operation on matrices of several dimensions $t_1, t_2, \ldots, t_k$) to build a more accurate performance model $f(t,x)$ of the true runtime $y(t,x)$ to use for tuning (by choosing x to minimize $f(t_i,x)$, for any t_i). In the common case that the performance varies reasonably smoothly as a function of both t and x, using all the available data to build $f(t,x)$ can make it more accurate, and so better for tuning. Once a predicted optimal x_{opt} is chosen for a particular t_i, the application can be run to measure the actual performance for the values $t = t_i$ and $x = x_{opt}$, and this data is used to update the model $f(t,x)$ to make it more accurate; this process of predicting x_{opt}, measuring the actual performance, and updating the model with the new x sample, can be repeated a user-selected number of times.

TLA goes beyond MLA by using the performance model to predict (and optimize) the performance for tasks (eg values of t) for which no true performance data has been collected. Here we assume that a MLA performance model without using the performance data of these new tasks has been built. The simplest way to do this is to use the same Gaussian Process approach used above to build a model of $x_{opt}(t) = \arg\min_x f(t,x)$, using the known (predicted) values of $\arg\min_x f(t_i,x)$ from MLA. This requires no additional collection of actual performance data. Alternatively, TLA can collect extra performance data for the new tasks, update (rather than rebuild) the MLA model, and predict optimal x for the new tasks (future work).

To simplify the above description, we chose a simple example with one parameter (t) needed to describe the task, and one tuning parameter (x). In practice, there may be many parameters needed to describe the task, and many tuning parameters, which could be of different types (real, integer, or "categorical", i.e. a list of discrete possibilities, such as choices of algorithms). Therefore, our interface (described in detail below) needs to accommodate all these possibilities.

We also need to accommodate constraints on the tuning parameters. For example, the user may require that the number of processors used (a tuning parameter) be less than or equal to an upper bound they supply, or that the product of the number of "processor rows" and the number of "processor columns" be less than the same upper bound. Our interface accommodates these and other possibilities.

If the measured performance data consists of multiple quantities, such as (runtime, accuracy), then the user may want to perform multi-objective optimization. For example, in the case of runtime and accuracy, which are likely to tradeoff against one another (faster runtime leading to worse accuracy), then the user may want to compute the Pareto front of these two quantities. Again, our interface accommodates this and other possibilities.

It is often the case that runtime data will be collected over time by one or more users running many different cases of a popular application, so we want to take advantage of all this data to improve the accuracy of our performance model. Our interface allows historical performance data to be stored in files, and reused later. We will improve this interface in future work.

To make it easier for users to try different autotuners, since several are available or under active development, our interface allows the user to invoke them as well. So far, OpenTuner [9] and ytopt [15] are supported.

It is sometimes the case that a user has a possibly coarse performance model that can be used to help prune the search space, even if it is not very accurate. Our interface allows the user to submit multiple models, for example just counting arithmetic operations, words communicated, etc., all of which are incorporated in the model GPTune builds, and can help accelerate tuning.

To simplify notation, in the rest of this manual 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, like x above, that the user wants to optimize; again there are generally multiple parameters to be tuned. 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 time.

Table 1 summarizes the notations used in the manual. As an illustrative example, the *QR factorization* routine of *ScaLAPACK* [5], denoted as *PDGEQRF*, is used as an application code to be tuned assuming a fixed core count $nprocmax$. So we list its respective parameters in this table. Note that only independent task and tuning parameters are listed here. Other parameters, such as the number of threads $nthreads$ (used in BLAS) and number of column processes q, can be calculated as $nthreads = \lfloor nprocmax/nproc \rfloor$ and $q = \lfloor nproc/p \rfloor$. Note that we can easily enforce that $nproc$ be a multiple of a certain integer (say 4) by using a modified parameter $nproc'$ such that $nproc = 4 \cdot nproc'$.

At a higher level, we also need to distinguish two independent sets of parameters:

- The parameters associated with the application codes, as described above, i.e. task parameters (as input to GPTune) and tuning parameters (as output from GPTune). Section 4.3 shows the API for the user to define these parameters.
- The parameters associated with the tuner itself, referred to as "GPTune parameters". These are related to the various tuning algorithms, such as MLA and TLA. Sections 4.10, 4.11 and 4.12 show the APIs for the user to choose the GPTune parameters. The settings of these parameters can affect the speed and accuracy of the tuning algorithms.

We have parallelized the most time-consuming part of the GPTune algorithms. Section 3.2 describes the parallel programming model on multicore nodes using MPI and OpenMP. The user can define the parallel computer configuration (nodes, cores) for GPTune's internal computation. The application to be tuned may also be executed in parallel, but GPTune and the objective function may use different numbers of nodes and cores. Section 4.5 shows the API for the user to specify the computer resources.

The rest of this user manual is organized as follows. Section 2 says how to install the system. Section 3 describes the user interface. Section 4 describes the underlying autotuning algorithms. Section 5 shows examples of autotuning real applications. Section 6 presents some performance results.

	Symbol	Interpretation
		General notations
	$\mathbb{IS}$	Task Parameter **I**nput **S**pace
	$\mathbb{PS}$	Tuning **P**arameter **S**pace (parameter configurations)
	$\mathbb{OS}$	**O**utput **S**pace (e.g., runtime)
	$\mathbb{MS}$	performance **M**odel **S**pace (e.g., flop count)
	α	dimension of $\mathbb{IS}$
	β	dimension of $\mathbb{PS}$
	γ	dimension of $\mathbb{OS}$
	$\tilde{\gamma}$	dimension of $\mathbb{MS}$
	δ (NI)	number of tasks
	ϵ (NS, NS1)	number of samples per task
	$T \in \mathbb{IS}^{\delta}$	array of tasks selected from sampling
	$X \in \mathbb{PS}^{\delta\times\epsilon}$	array of samples (parameter configurations)
	$Y \in \mathbb{OS}^{\delta\times\epsilon}$	array of output results (e.g. runtime)
		Example: parameters for ScaLAPACK *PDGEQRF* notations
Task	m	number of matrix rows
	n	number of matrix columns
Tuning	mb	row block size
	nb	column block size
	$nproc$	number of MPI processes
	p	number of row processes

Table 1: Notations. The symbols in the parentheses denote code notations.

2 Installation

GPTune is implemented in Python and C, and it depends on several Python, Fortran and C packages. Most of them can be installed with one line; the rest requires manual installation. Before the installation, the following software environment is needed (with the minimum version number in the parentheses): **gcc**(7.4.0), **openmpi**(4.0.1), **python**(3.7), **scalapack**(2.0.2), **git**, **cmake**, **blas**, **lapack**, (**strumpack** in the future). The number in parentheses denotes the required minimum version number. Note that openmpi, python and ScaLAPACK should use the same gcc version. It might be necessary to build them from source. We suggest the readers take a look at .travis.yml in the GPTune root folder for the complete installation from scratch.

GPTune can be obtained from its github repository:

```
export GPROOT=root-folder-of-GPTune
cd $GPROOT
git clone https://github.com/xiaoyeli/GPTune.git
```

Setting the following environment variables makes it convenient for consequent installation.

```
export PYTHONWARNINGS=ignore
export CCC=path-to-the-mpicc-wrapper
export CCCPP=path-to-the-mpic++-wrapper
export FTN=path-to-the-mpif90-wrapper
export RUN=path-to-the-mpirun-wrapper
export BLAS_LIB=path-to-the-blas-lib
export LAPACK_LIB=path-to-the-lapack-lib
export SCALAPACK_LIB=path-to-the-scalapack-lib
```

2.1 Manual installation

The following packages need to be installed manually.

2.1.1 GPTune C code

```
cd $GPROOT
mkdir -p build
cd build
cmake .. \
-DBUILD_SHARED_LIBS=ON \
-DCMAKE_CXX_COMPILER=$CCCPP \
-DCMAKE_C_COMPILER=$CCC \
-DCMAKE_Fortran_COMPILER=$FTN \
-DTPL_BLAS_LIBRARIES=$BLAS_LIB\
-DTPL_LAPACK_LIBRARIES=$LAPACK_LIB \
-DTPL_SCALAPACK_LIBRARIES=$SCALAPACK_LIB
make
cp lib_gptuneclcm.so ../.
cp pdqrdriver ../
```

2.1.2 mpi4py

```
cd $GPROOT
rm -rf mpi4py
git clone https://github.com/mpi4py/mpi4py.git
cd mpi4py/
python setup.py build --mpicc="$CCC -shared"
python setup.py install
export PYTHONPATH=$YTHONPATH:$GPROOT/mpi4py/
```

2.1.3 autotune

```
cd $GPROOT
rm -rf autotune
git clone https://github.com/ytopt-team/autotune.git
cd autotune/
pip install --user -e .
export PYTHONPATH=$YTHONPATH:$GPROOT/autotune/
```

2.1.4 GPTune Examples

```
cd $GPROOT/examples/
git clone https://github.com/xiaoyeli/superlu_dist.git
cd superlu_dist
mkdir -p build
cd build
```

```
cmake .. \
-DCMAKE_CXX_FLAGS="-Ofast -std=c++11 -DAdd_ -DRELEASE" \
-DCMAKE_C_FLAGS="-std=c11 -DPRNTlevel=0 -DPROFlevel=0 -DDEBUGlevel=0" \
-DBUILD_SHARED_LIBS=OFF \
-DCMAKE_CXX_COMPILER=$CCCPP \
-DCMAKE_C_COMPILER=$CCC \
-DCMAKE_Fortran_COMPILER=$FTN \
-DTPL_BLAS_LIBRARIES=$BLAS_LIB \
-DTPL_LAPACK_LIBRARIES=$LAPACK_LIB \
-DTPL_PARMETIS_INCLUDE_DIRS=path-to-parmetis-include \
-DTPL_PARMETIS_LIBRARIES=path-to-parmetis-lib
make pddrive_spawn
```

3 GPTune Implementation

In this section, we describe how GPTune is implemented, with sufficient details of the tuning algorithm so that the user knows how to set GPTune's learning algorithm parameters (see Sections 4.10 and 4.11) and other options (see Section 4.12). We refer the users to our technical paper [14] for a thorough exposition of the methodology.

3.1 Algorithms

We first describe the algorithm for single-objective autotuning, then describe the algorithm for multi-objective autotuning.

3.1.1 Single-objective autotuning

Recall that we seek to optimize some objective function with a set of tunable parameters for best performance, in the form of $\arg\min_x y(t, x)$, where $t \in \mathbb{IS}$ is an input task and $x \in \mathbb{PS}$ is a tuning parameter configuration. $\mathbb{IS}$ is the *Task Parameter Input Space* containing all the input problems that the application may encounter. Note: 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.

We can evaluate the objective function pointwise at a tuning parameter configuration (i.e., run and measure the application on a parallel machine), but the function does not have an easy closed form nor easy-to-compute gradients. Moreover it is expensive to perform a function evaluation. Given these characteristics, we choose to use the Bayesian optimization method (also known as Efficient Global Optimization (EGO) [10]), in which a prior belief model representing the assumptions on the objective function is chosen, and a posterior is built from it so as to maximize the likelihood of some probability distribution function based on the sampled objective function values.

Compared to the other autotuning efforts, one of our innovations is to use *multitask learning* to build a more accurate predictive model. Multitask learning consists of learning several tasks simultaneously (eg running a ScaLAPACK routine with different matrix dimensions) while sharing common knowledge between them in order to improve the prediction accuracy of each task and/or speed up the training process. We call this framework multitask learning autotuning (MLA).

Specifically, the MLA learning process consists of the following phases, where we also define GPTune parameters, or provide names for other quantities that appear in the GPTune interface.

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 $X = [X_1; X_2; \ldots; X_\delta]$ of tuning parameter configurations for every task. Let NS denote a prescribed total number of function evaluations per task. The number of initial samples is set to $\epsilon = \mathrm{NS1}$, with $\mathrm{NS1} = \lceil \mathrm{NS}/2 \rceil$ by default. 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.

   GPTune parameters. NI: number of tasks (i.e., δ). Note that we use NI as the code notation and δ as the algorithm notation. Igiven: (optional) list of user specified task parameters. NS: total number of function evaluations per task. NS1: (optional, default value given above) number of initial function evaluations. The other related GPTune parameters are: sample_class, sample_algo, sample_max1_iter, which are described in Section 4.12.

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 be able to better predict them all. To this end, we use the *Linear Coregionalization Model* (LCM), which is a generalization of *Gaussian Process* (GP) in the multi-output setting. 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 $\Sigma(x, x') = k(x, x')$, where k is a positive definite kernel function. The idea is that if x and x' are deemed similar by the kernel, 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 practice, $\mu(x)$ is 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 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:

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

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

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

where $\delta_{i,j}$ is the Kronecker delta function. The learning task in the nth MLA iteration is to find the best hyperparameters of the model, such as the hyperparameters σ_q, l_i^q in the Gaussian kernel Eq.(3) and coefficients $a_{i,q}, d_i$ 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) [13] through the Python package scikit-Optimize [3]. Note that the log-likelihood function is usually highly nonconvex, so local optimization may not converge to the/a global optimum. The modeling phase extends the Python package GPy [8] to enable distributed-memory parallel modeling.

GPTune parameters. model_latent: number of latent functions (i.e., Q). By default $Q = \delta$. The other relevant options are model_restarts, model_max_iters, etc. See Section 4.12 for details.

Algorithm 1 Bayesian optimization-based single-objective MLA

```
1: Sampling phase: Compute $y(t_i, x)$, $i \leq \delta$ at NS1 initial random tuning parameter configurations for $\delta$ selected tasks. Set $\epsilon =$ NS1.
2: while $\epsilon \leq$ NS do
3:     Modeling phase: Update the hyperparameters in the LCM model of $y(t_i, x)$, $i \leq \delta$ using all available data.
4:     Search phase: Search for an optimizer $x_i^*$ for the EI of task $t_i$, $i \leq \delta$. Let $X^* = [x_1^*; x_2^*; \ldots; x_\delta^*]$.
5:     Compute $y(t_i, x)$, $i \leq \delta$ at the new tuning parameter configurations $X^*$.
6:     $\epsilon \leftarrow \epsilon + 1$.
7: end while
8: Return the optimum tuning parameter configurations and objective function values for each task.
```

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 (prediction confidence) $\sigma_*^2 = [\sigma_1^{*2}; \sigma_2^{*2}; \ldots; \sigma_\delta^{*2}]$ as:

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

$$\sigma_*^2 = \Sigma(X^*, X^*) - \Sigma(X^*, X)\Sigma(X, X)^{-1}\Sigma(X^*, X)^T \tag{6}$$

Note that the posterior variance is equal to the prior covariance minus a term that corresponds to the variance removed by observing X [7]. The mean and variance can be used to construct the Expected Improvement (EI) acquisition function, which can be maximized in order to choose a new point X^* for function evaluation (additional sampling). We use evolutionary algorithms provided by the python package PyGMO [2] to optimize the EI. PyGMO supports many optimization algorithms. By default, we use Particle Swarm Optimization (PSO) algorithm [11].

With one additional function evaluation, we increment ϵ by 1 and move to next MLA iteration for model improvement until ϵ reaches a prescribed sample count NS (i.e., a prescribed budget of function evaluations). This iterative process is summarized as Algorithm 1.

GPTune parameters. search_algo: evolutionary algorithms supported by PyGMO. By default search_algo is 'pso'. The other relevant options are search_pop_size, search_gen, search_evolve, search_max_iters, etc. See Section 4.12 for details.

3.1.2 Multi-objective autotuning

The MLA algorithm described in Section 3.1.1 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) [6] to search for k new tuning parameter configurations in each iteration. The sorting is based on the Pareto dominance and Crowding distance [6].

Algorithm 2 Bayesian optimization-based multi-objective MLA

```
1: Sampling phase: Compute $y^s(t_i, x)$, $i \leq \delta$, $s \leq \gamma$ at NS1 initial random tuning parameter
   configurations for $\delta$ selected tasks. Set $\epsilon$ = NS1.
2: while $\epsilon \leq$ NS 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 for the 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:     $\epsilon \leftarrow \epsilon + k$.
7: end while
8: Return the optimum tuning parameter configurations and objective function values for each
   task.
```

GPTune parameters. search_algo: evolutionary algorithms supported by PyGMO. By default search_algo is 'nsga2' (i.e., the NSGA-II algorithm). search_more_samples: the number of additional samples per iteration (i.e., k). The other relevant options are search_pop_size, search_gen, search_evolve, search_max_iters, etc. See Section 4.12 for details.

3.1.3 Incorporation of performance models

A performance model refers to an analytical formula or inexpensive application run 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 task $\delta = 1$ and 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 $\tilde{y}(x)$ denote the results of the performance models for parameter configuration x. Without the performance model, entries of the LCM kernel matrix represents the nonlinear inner products between points x and x' in the feature space $\mathbb{PS}$ of dimension β. One can use the values $\tilde{y}(x)$ as the extra features to construct an enriched feature space of dimension $\beta + \tilde{\gamma}$ consisting points $[x, \tilde{y}(x)]$. 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 $[x^*, \tilde{y}(x^*)]$.

Note that GPTune internally represents the tuning parameters x as real numbers in the unit hypercube $[0,1]^\beta$. Therefore, it is beneficial that the $\tilde{y}(x)$ can be normalized by the users to $[0,1]^{\tilde{\gamma}}$. For example, if the users know in advance that $\tilde{y}(x) \in [y_l, y_u]$ assuming $\tilde{\gamma} = 1$, they can instead return the output of the model as $\tilde{y}(x) \to (\tilde{y}(x) - y_l)/(y_u - y_l)$. We recommend the users to perform such conversion even when y_l and y_u are approximate numbers, as this will help GPTune build a more accurate LCM model.

3.1.4 Transfer learning

Transfer learning autotuning (TLA) focuses on the case where no true performance data has been collected for a specified task t^*, but rather performance data and models have been built for different (but related) tasks $t_i, i \le \delta$. The simplest way to do this is to use the Gaussian Process approach to build a model of $x_{opt}(t) = \arg\min_x f(t, x)$, using the known (predicted) values of $\arg\min_x f(t_i, x)$ from MLA. This requires no additional collection of actual performance data. GPTune uses Algorithm 1 with $\delta = 1$ to build this model.

Alternatively, the model could be iteratively used to predict an optimal x, measure the performance, and update the model. This is future work.

3.2 Parallel implementations

GPTune supports both shared-memory and distributed-memory parallelism through dynamic thread and process management. Most parts of GPTune are implemented with Python3. The shared-memory parallelism is supported through OpenMP threading or the subclass ThreadPoolExecutor from the Python module *concurrent.futures*. The distributed-memory parallelism is supported through the Python package mpi4py [1] and will be explained in detail.

3.2.1 Dynamic process management

In our design, only one MPI process can execute the GPTune driver, but it can also dynamically create 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.

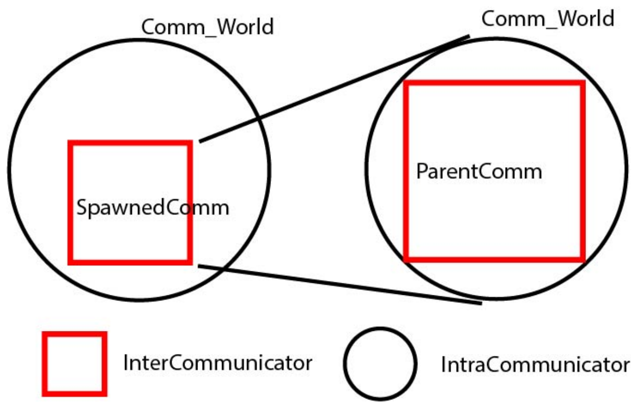

Figure 1: GPTune parallel programming model.

A typical spawning call on the master process is

```
SpawnedComm = mpi4py.MPI.COMM_SELF.Spawn(executable, args, maxprocs,info)
```

Here, "maxprocs" is the number of new processes to be created, "executable" is the program to be executed by the workers, "args" are the command line arguments to be passed to the program, "info" are the environment variables to be passed to the program, and "SpawnedComm" is the inter-communicator with the master as the local group.

A typical setup on the worker processes is

```
ParentComm = mpi4py.MPI.Comm.Get_parent()
```

Here "ParentComm" is the inter-communicator with the workers as the local group. Note that Get_parent can be called from C, C++ or Fortran application codes with a slightly different syntax.

In what follows, we describe the shared-memory and distributed-memory parallelism in the objective function evaluation, modeling phase (in MLA), and search phase (in MLA) separately. Parallel implementation of TLA is future work.

3.2.2 Objective function evaluation

The users of GPTune will need to provide the objective function (see the definition in Section 4.1) that uses the task and tuning parameters to execute the application code. There are two levels of parallelism supported: running a single objective function evaluation with given MPI and thread counts, and running multiple objective function evaluations in parallel.

Running one function evaluation in parallel. For a single parallel objective function evaluation, the MPI count can be passed to the application code using the argument "maxprocs", the thread count can be passed using the argument "info" as

```
info = mpi4py.MPI.Info.Create()
info.Set('env', 'OMP_NUM_THREADS=%d\n' %(nthreads))
```

where "nthreads" is the number of threads possibly contained in the task or tuning parameters. Depending on how the application code is implemented, one can pass the parameters using command line ("args"), environment variables ("info"), or an input file stored on disc.

To collect the returning value(s) from the workers, one can choose to read from the individual log file per function execution (see the example of ScaLAPACK QR in Section 5.1) or communicate using the inter-communicators (see the example of SuperLU_Dist in Section 5.2). For example, assuming the application code is written in C, the two suggested ways of communicating data are:
Using log file: On the master side, write the (Python) objective function (see Section 4.1) in the following fashion:

```
def objectives(point):
# extract task and tuning parameters from "point", and use them to define
    enviroment variables (info), command line arguments (args), MPI counts (
    maxprocs) and input files if needed
info = mpi4py.MPI.Info.Create() # enviroment variables
envstr= 'env1=%d\n' %(ev1)
envstr+= 'env2=%d\n' %(ev2)
info.Set('env',envstr)
args = ['-a1', '%s'%(av1), '-a2', '%s'%(av2)] # command line arguments
create the input file needed by executable  # input file
SpawnedComm = mpi4py.MPI.COMM_SELF.Spawn(executable, args, maxprocs,info) # call
    executable
SpawnedComm.Disconnect() # destroy inter-communicator
read objective values from the log file into res  # output file
return res
```

On the workder side, the C function looks like the following:

```
int main(int argc, char *argv[]){
MPI_Init( &argc, &argv );
MPI_Comm parent; MPI_Comm_get_parent(&parent); /* Get the inter communicator. */
.../* Read the parameters from environment variable, command lines and/or input
    files, compute the objective function and dump it into a log file. */
MPI_Comm_disconnect(&parent); /* Disconnect the inter communicator. */
MPI_Finalize();}
```

Using inter-communicator: For example, the master can collect data using MPI_Reduce as

```
SpawnedComm.Reduce(sendbuf=None,recvbuf=data,op=MPI_OP,root=mpi4py.MPI.ROOT)
```

and the workers send the data by

```
ParentComm.Reduce(sendbuf=data,recvbuf=None,op=MPI_OP,root=0)
```

Here, “data” is the returning value, “MPI_OP” is the MPI reduce operation. Again, the syntax can be different on the workers depending on the programming language of the application code. One can also consider using MPI_Send and MPI_Recv for passing the data back to the master, please refer to the mpi4py documentation for more details. One can consider modifying the following example: On the master side, the (Python) objective function (see Section 4.1) looks like the following:

```
def objectives(point):
# extract task and tuning parameters from "point", and use them to define
    enviroment variables (info), command line arguments (args), MPI counts (
    maxprocs) and input files if needed
info = mpi4py.MPI.Info.Create() # enviroment variables
envstr= 'env1=%d\n' %(ev1)
envstr+= 'env2=%d\n' %(ev2)
info.Set('env',envstr)
args = ['-a1', '%s'%(av1), '-a2', '%s'%(av2)] # command line arguments
create the input file needed by executable  # input file
SpawnedComm = mpi4py.MPI.COMM_SELF.Spawn(executable, args, maxprocs,info) # call
    executable
SpawnedComm.Reduce(sendbuf=None,recvbuf=data,op=MPI_OP,root=mpi4py.MPI.ROOT) # use
     MPI_Reduce for collect the objectives
SpawnedComm.Disconnect() # destroy inter-communicator
return res
```

On the workder side, the C function looks like the following:

```
int main(int argc, char *argv[]){
MPI_Init( &argc, &argv );
MPI_Comm parent; MPI_Comm_get_parent(&parent); /* Get the inter communicator. */
.../* Read the parameters from environment variable, command lines and/or input
    files, and compute the objective function. */
MPI_Reduce(result, MPI_BOTTOM, 1, MPI_FLOAT,MPI_MAX, 0, parent); # return the
    values "result" to the master
MPI_Comm_disconnect(&parent); /* Disconnect the inter communicator. */
MPI_Finalize();}
```

Running multiple function evaluations in parallel For applications that require a small to modest core count for each objective function evaluation, it’s beneficial to even perform multiple function evaluations simultaneously. GPTune uses MPI spawning and ThreadPoolExecutor to support distributed- and shared-memory parallelism over number of function evaluations. Note that the users need to use this feature with caution: one needs to make sure the output and input files (when they exist) from different function evaluations do not interfere with each other. This can be done by e.g., assigning each function evaluation a unique file or directory name.

GPTune parameters. objective_nprocmax: maximum core counts for each function evaluation. objective_evaluation_parallelism: whether to use parallelism over different function evaluations. objective_multisample_processes: when both objective_evaluation_parallelism and distributed_memory_parallelism are set to True, this parameter denotes the number of process groups, each responsible for one function evaluation. objective_multisample_threads: when both objective_evaluation_parallelism and shared_memory_parallelism are set to True, this parameter denotes the number of threads processes for parallelizing over the number of function evaluations. See Section 4.12 for details.

3.2.3 Modeling phase of MLA

The modeling phase, described in Section 3.1.1, uses the L-BFGS algorithm to find a set of LCM hyperparameters that minimizes 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 minimum log-likelihood. GPTune then chooses the set of hyperparameters that yields the best log-likelihood and finishes the modeling phase.

The current implementation supports two levels of parallelism in this phase: (1) The number of n_{start} random starts and corresponding L-BFGS optimization are distributed over user specified number of threads or MPI processes. Note that the shared-memory parallelism and distributed-memory parallelism (over n_{start}) are supported mutually exclusively. GPTune uses MPI spawning to support the distributed-memory parallelism for the random starts. (2) For each L-BFGS optimization, the factorization of the covariance matrix is parallelized over user specified number of threads and MPI processes, the formation of the covariance matrix is parallelized over user specified number of threads. GPTune uses MPI spawning for distributed-memory parallelization of the covariance matrix.

GPTune parameters. model_restarts: number of random starts of the hyperparameters (i.e., n_{start}). distributed_memory_parallelism: whether to use distributed-memory parallelism over the random starts. model_restart_processes: number of MPI processes for parallelizing over n_{start}. shared_memory_parallelism: whether to use shared-memory parallelism over the random starts. model_restart_threads: number of threads for parallelizing over n_{start}. model_processes: number of MPI processes for parallelizing factorization of the covariance matrix. model_threads: number of OpenMP threads for parallelizing formation and factorization of the covariance matrix. See Section 4.12 for details.

3.2.4 Search phase of MLA

The search phase uses evolutionary algorithms in PyGMO to search for the next sample point in each task (see Section 3.1.1 for details). The GPTune implementation supports two levels of parallelism: (1) The multi-task search can be parallelized over the δ tasks using user-specified number of threads or MPI processes. Note that the shared-memory parallelism and distributed-memory parallelism (over δ) are supported mutually exclusively. GPTune uses MPI spawning to support the distributed-memory parallelism over the tasks. (2) For each task, the evolutionary algorithm can be parallelized using user-specified number of threads.

GPTune parameters. distributed_memory_parallelism: whether to use distributed-memory parallelism over the δ tasks. search_multitask_processes: number of MPI processes for parallelizing over δ. shared_memory_parallelism: whether to use shared-memory parallelism over δ. search_multitask_threads: number of threads for parallelizing over δ. search_threads: number of threads used in PyGMO. See Section 4.12 for details.

3.3 Auto installation

The following python packages listed in requirement.txt can be installed automatically with pip: **numpy, scikit-learn, scipy, matplotlib, GPy, openturns, lhsmdu, pygmo, ipyparallel, opentuner, hpbandster, scikit-optimize**.

```
pip install --upgrade --user -r requirements.txt
```

4 User Interface

This section describes the essential APIs for a GPTune driver. For illustration purposes, we will describe the interfaces in the context of tuning the runtime of the parallel QR factorization routine *PDGEQRF* from the ScaLAPACK package [5]. For the QR factorization time of a matrix, we can consider the matrix dimensions (m, n) to be the task parameters. Let row block size, column block size, number of MPI processes, number of row processes, denoted by $(mb, nb, nproc, p)$ be the parameters to be tuned assuming a fixed total number of cores $nprocmax$. Note that other arguments affecting the runtime such as the number of column processes and threads per process can be derived from these arguments. Finally let (r) be the QR runtime with fixed task parameters and tuning parameters.

4.1 Define the objective function to be minimized

```
# define the multi or single objective function
def objectives(point):
  # extract task and tuning parameters from "point", call the application code,
    and collect the results in "res".
  return res
```

- **point** [Dict] (input): A dictionary containing the task parameter and tuning parameter arguments of the objective function. point[“I_j”] is the value of the task parameter argument named “I_j”, point[“P_j”] is the value of the parameter argument named “P_j”, see Section 4.3 for their definitions.

- **res** [numpy array, shape=(1,γ)] (output): Array containing the objective function value, γ denotes dimension of the output space $\mathbb{OS}$.

QR EXAMPLE:
point[“m”], point[“n”], point[“mb”], point[“nb”], point[“nproc”], point[“p”] are the task parameter and tuning parameter arguments for *PDGEQRF*. The user is responsible for writing a driver code that calls *PDGEQRF* and returns the runtime in **res** with $\gamma = 1$, e.g., using a MPI spawning approach. See Section 5 for more details.

4.2 Define the performance models

```
# define the coarse performance models
def models(point):
# extract task and tuning parameters from "point", call the performance models,
    and collect the results in "res".
return res
```

- **point** [Dict] (input): A dictionary containing the task parameter and tuning parameter arguments of the objective function. point[“I_j”] is the value of the task parameter argument named “I_j”, point[“P_j”] is the value of the parameter argument named “P_j”, see Section 4.3 for their definitions. Note that the input “point” is exactly the same as that for “objectives”.

- **res** [numpy array, shape=(1,$\tilde{\gamma}$)] (output): Array containing the performance models outputs. Note that the number of performance models $\tilde{\gamma}$ can be different from γ .

QR EXAMPLE:
point["m"], point["n"], point["mb"], point["nb"], point["nproc"], point["p"] are the task parameter and tuning parameter arguments for *PDGEQRF*. The user can use a simple performance model **res**= $[mn^2/nproc/PF]$, with PF denotes the peak flop rate of the machine.

4.3 Define the tuning parameter, task parameter and output spaces

```
IS = Space([I_1,I_2,...,I_α]): # task parameter space with instances of supported spaces
    , α is the dimension of the task parameter space
PS = Space([P_1,P_2,...,P_β]): # tuning parameter space with instances of supported
    spaces, β is the dimension of the tuning parameter space
OS = Space([O_1,O_2,...,O_γ]): # output space with instances of supported spaces, γ is
    the dimension of the output space. Note that the return value of the Callable "
    objectives" is a point in this space.
constraints = {"cst1" : cst1, ...} # constraints for I_j and P_j
```

- ***I_j*, *P_j*** [Scikit-optimize spaces] (input): Please refer to `https://scikit-optimize.github.io/stable/modules/classes.html?highlight=space#module-skopt.space.space` for the scikit-optimize spaces. Supported spaces for I_j (and P_j) :
  I_j = Integer (low , high, transform="normalize", name="I_j"), here the "normalize" transform converts the integers in the range [low,high] to real numbers in $[0, 1]$, and vice versa. Please refer to `https://scikit-optimize.github.io/stable/modules/generated/skopt.space.transformers.Normalize.html` for more details about the transform. Note that it is required that low<high.
  I_j = Real(low, high, transform="normalize", name="I_j"), here the "normalize" transform converts real numbers in the range [low,high] to $[0, 1]$. It is required that low<high.
  I_j = Categoricalnorm (categories, transform="onehot", name="I_j"). Note: Categoricalnorm is compatible with Categorical but with a modified transform "onehot" that converts the categorical data to real numbers in $[0, 1)$. Speficically, the transform first converts the categorical data to its onehot encoding which is a $1 \times \alpha$ binary array with only element of 1 (e.g. the kth element), then converts it to the real number $(k-1)/\alpha + 10^{-12}$. Conversely, the modified transform converts any number in $[(k-1)/\alpha, k/\alpha)$ to the onehot encoding with kth element being 1, representing the kth category.

- ***O_j*** [Scikit-optimize spaces] (input): Supported spaces:
  O_j = Real(low, high, name="O_j"). Note: if low and high are unknown, they can be set to float("-Inf") and float("Inf"), respectively.

- **cst1** [string] (input): define one constraint, e.g., as cst1 = "$I_1 + P_2 < 10$".

QR EXAMPLE:

```
m  = Integer (128 , 2000, transform="normalize", name="m") # row dimension
n  = Integer (128 , 2000, transform="normalize", name="n") # column dimension
IS = Space([m, n]) # task parameter space
mb  = Integer (1 , 512, transform="normalize", name="mb") # row block size
```

```python
nb  = Integer (1 , 512, transform="normalize", name="nb") # column block size
nproc  = Integer (1 , 4, transform="normalize", name="nproc") # number of MPIs
p  = Integer (1 , 4, transform="normalize", name="p") # number of row processes
PS = Space([mb, nb, nproc, p]) # tuning parameter space
r  = Real (float("-Inf") , float("Inf"), name="r") # runtime
OS = Space([r]) # output space
cst1 = "mb * p <= m"
cst2 = "nb * nproc <= n * p"
cst3 = "nproc >= p"
constraints = {"cst1" : cst1, "cst2" : cst2, "cst3" : cst3} # constraints for task
    parameters and tuning parameters
```

4.4 Define the tuning problem

```python
problem = TuningProblem(IS, PS, OS, objectives, constraints, models)
# define the tuning problem from the spaces, objective function and constraints
```

- **IS** [Space] (input): The task parameter space defining the objective function
- **PS** [Space] (input): The tuning parameter space defining the tuning parameters of the objective function
- **OS** [Space] (input): The output space defining the output of the objective function
- **objectives** [Callable] (input): The objective function to be minimized
- **constraints** [Dict] (input): The constraints for the task parameters and tuning parameters
- **models** [Callable] (input): The avaiable performance models. This can be None if no model is available.
- **problem** [Class] (output): The tuning problem

4.5 Define the computation resource

```python
computer = Computer(nodes, cores, hosts) # define the computation resource used in
    GPTune's internal computation. Note: the same amount of resource can be used
   for invoking the objective functions as the function evaluation and the tuner's
    internal computation do not run parallel to each other. See Section 5 for
   examples.
```

- **nodes** [Int] (input): The minimum number of MPI processes used for GPTune different phases
- **cores** [Int] (input): The maximum number of threads per MPI process used for GPTune different phases
- **hosts** [Collection] (input): The list of hostnames
- **computer** [Class] (output): The computer class

4.6 Define and validate the options

```
options = Options() # define the default options
```

- **options** [Class] (output): The options class. See GPTune/options.py and Section 4.12 for all the options.

```
options['item'] = val # set the option named 'item' to val.
```

```
options.validate(computer) # check the options specified by the user
```

- **computer** [Class] (input): The computer class

4.7 Create the data class for storing samples of the spaces

```
data = Data(problem) # define the data class
```

- **problem** [Class] (input): The tuning problem class
- **data** [Class] (output): The data class to be used for storing sampled tasks, tuning parameters and outputs.
  data.I [list of lists]: Each entry is a list of length α representing one task sample.
  data.P [list of [list of lists]]: Each entry corresponds to one task sample. For each entry (list of lists), each entry is a list of length β representing one tuning parameter configuration.
  data.O [list of numpy arrays]: Each entry is a numpy array of shape $(,\gamma)$ representing objective function values for one task sample. One row of each array represents the objective function values for one tuning parameter configuration.
  On return, data.I=None, data.P=None, data.O=None. See Section 4.10 for details.

4.8 Use historical data from previous MLA runs

```
data = pickle.load(open(filename, 'rb'))
```

- **filename** [Pickled data] (input): The binary file storing the data class from previous MLA runs.
- **data** [Class] (output): The data class to be updated for the new MLA run.
  GPTune supports the usage of historical MLA data in a checkpoint fashion. Rather than creating an empty data class using the interface of Section 4.7, this function returns a data class with len(data.I)$\neq 0$, len(data.P[0])$\neq 0$, and len(data.O[0])$\neq 0$.

4.9 Initialize GPTune

```python
gptune = GPTune(problem, computer, data, options)
# initialize the tuner from the meda data
```

- **problem** [Class] (input): The tuning problem
- **computer** [Class] (input): The computer
- **data** [Class] (input): The data class. If any of data.I, data.P and data.O is None, the tuner will generate random samples for it later.
- **options** [Class] (input): The options class
- **gptune** [Class] (input): The tuner class that registers problem, computer, data and options as gptune.problem, gptune.computer, gptune.data, and gptune.options, respectively.

4.10 Call multi-task learning algorithm (MLA)

```python
(data, models,stats) = gptune.MLA(NS, NI, Igiven, NS1)
# build the MLA models and search for the optimum tuning parameters on each task
```

- **gptune** [Class]: The tuner
- **NS** [Int] (input): Number of total samples per task to be returned. Note that the tuner returns immediately if the number of samples in historical data is more than NS.
- **NI** [Int] (input): Number of tasks to be modeled (i.e., NI=δ) (Note that in the MLA interface NI should match the number of tasks in the historical data (if present). If one needs to add new tasks to the historical data, use the TLA interface in Section 4.11 instead.)
- **Igiven** [list of lists] (input): A list of prescribed task parameters. Note that Igiven should match the list of task parameters in the historical data (if present).
- **NS1** [Int] (input):If no historical data is presesnt, the tuner generates NS1 initial random tuning parameter samples before starting the adaptive model refinement.
- **data** [Class] (output): The data class containing all the task, tuning parameter and output sampled by the tuner. len(data.I)=NI, len(data.P)=NI, len(data.O)=NI, and len(data.P[0])=NS.
- **models** [list of Class] (output): Each entry represents the trained LCM model for one objective in the adaptive model refinement.
- **stats** [Dict] (output): Memory and CPU profiles generated by the tuner

4.11 Call transfer learning algorithm (TLA)

```
(aprxopts,objval,stats) = gptune.TLA(newtask, NS)
# Use existing data and MLA model on pre-tuned tasks (stored in gptune.data and
    gptune.models) and TLA to search for the optimum tuning parameters on each new
    task
```

- **gptune** [Class]: The tuner that encapsulates the previous tuning data and models.
- **newtask** [list of lists] (input): newtask consists of list of prescribed tasks to be tuned
- **NS** [Int] (input): Maximum number of objective function evaluations
- **aprxopts** [list of lists] (output): Each entry (list) of the list corresponds to the predicted best tuning parameters
- **objval** [list of numpy arrays] (output): Each entry of the list corresponds to objective function values using the predicted tuning parameters for one task
- **stats** [Dict] (output): Memory and CPU profiles generated by the tuner

4.12 GPTune options

The options affecting the efficiency of GPTune are listed below.

```
class Options(dict):
  def __init__(self, **kwargs):

    """ Options for GPTune """
    mpi_comm = None           # The mpi communiator that invokes gptune if mpi4py
    is installed
    distributed_memory_parallelism = False   # Using
    distributed_memory_parallelism for the modeling (one MPI per model restart) and
     search phase (one MPI per task)
    shared_memory_parallelism      = False   # Using shared_memory_parallelism for
     the modeling (one MPI per model restart) and search phase (one MPI per task)
    verbose = False     # Control the verbosity level
    oversubscribe = False     # Set this to True when the physical core count is
    less than computer.nodes*computer.cores and the --oversubscribe MPI runtime
    option is used

    """ Options for the function evaluation """
    objective_evaluation_parallelism   = False  # Using
    distributed_memory_parallelism or shared_memory_parallelism for evaluating
    multiple application instances in parallel
    objective_multisample_processes = None  # Number of MPIs each handling one
    application call
    objective_multisample_threads = None  # Number of threads each handling one
    application call
    objective_nprocmax = None # Maximum number of cores for each application call,
     default to computer.cores*computer.nodes-1

    """ Options for the sampling phase """
    sample_class = 'SampleLHSMDU' # Supported sample classes: 'SampleLHSMDU', '
    SampleOpenTURNS'
```

```python
    sample_algo = 'LHS-MDU' # Supported sample algorithms: 'LHS-MDU' --Latin hypercube sampling with multidimensional uniformity, 'MCS' --Monte Carlo Sampling
    sample_max_iter = 10**9  # Maximum number of iterations for generating random samples and testing the constraints

    """ Options for the modeling phase """
    model_threads = None  # Number of threads used for building one GP model in Model_LCM
    model_processes = None # Number of MPIs used for building one GP model in Model_LCM
    model_restarts = 1 # Number of random starts each building one initial GP model
    model_restart_processes = None  # Number of MPIs each handling one random start
    model_restart_threads = None   # Number of threads each handling one random start
    model_max_iters = 15000   # Number of maximum iterations for the optimizers
    model_latent = None # Number of latent functions for building one LCM model, defaults to number of tasks

    """ Options for the search phase """
    search_threads = None  # Number of threads in each thread group handling one task
    search_processes = 1  # Reserved option
    search_multitask_threads = None # Number of threads groups each handling one task
    search_multitask_processes = None # Number of MPIs each handling one task
    search_algo = 'pso' # Supported search algorithm in pygmo: single-objective: 'pso' -- particle swarm, 'cmaes' -- covariance matrix adaptation evolution. multi-objective 'nsga2' -- Non-dominated Sorting GA, 'nspso' -- Non-dominated Sorting PSO, 'maco' -- Multi-objective Hypervolume-based ACO, 'moead' -- Multi-objective EA vith Decomposition
    search_pop_size = 1000 # Population size in pgymo
    search_gen = 1000  # Number of evolution generations in pgymo
    search_evolve = 10  # Number of times migration in pgymo
    search_max_iters = 10  # Max number of searches to get results respecting the constraints
    search_more_samples = 1  # Maximum number of points selected using a multi-objective search algorithm
```

Listing 1: Default GPTune options.

5 Example code

Use ScaLAPACK QR as a walk-through example. Can insert a performance model.

Description of how to use OpenTunner or Hyperbanster

5.1 ScaLAPACK QR

As mentioned in Section 4, we can use (m, n) to define a task, then $(mb, nb, nproc, p)$ to define the tuning parameters. Here we use simplified codes to illustrate a few typical use cases with GPTune. Please refer to GPTune/examples/scalapack*.py, GPTune/examples/scalapack-driver/spt/pdqrdriver.py, and GPTune/examples/scalapack-driver/src/pdqrdriver.f for the complete working codes.

5.1.1 MLA in a single run

This example builds a LCM model of the QR driver for two user specified tasks [[400,500],[800,600]]. The Python driver will dump the task parameter and tuning parameters into an input file named ‘‘GPTune/examples/scalapack-driver/exp/MACHINE_NAME/TUNER_NAME/JOBID/QR.in”, invoke the Fortran application code pdqrdriver.f, then read the return values from an output file named ‘‘GPTune/examples/scalapack-driver/exp/MACHINE_NAME/TUNER_NAME/JOBID/QR.out”. The variables “MACHINE_NAME”, “TUNER_NAME”, and “JOBID” can be defined by the user.

In terms of computation resource, “nodes=4” and “cores=8” are used for GPTune’s several phases; “nprocmax = nodes*cores-1” and “nprocmin = nodes” are used to define MPI count for the application code (pdqrdriver.f). Note that nprocmin and nprocmax are used as the “low” and “high” arguments for the Scikit-optimize space “Integer” (see line 104 of Listing 2), one needs to ensure nprocmin<npromax. As we use MPI spawn to invoke the application, one process is reserved as the spawning process.

Note that to reduce the runtime noise, the Python driver will execute the same task and tuning parameter configuration three times and return the minimum runtime as the function value.

```

''' Pass the inputs and parameters from Python to the Fortran driver using files RUNDIR/QR.in, note that the the inputs and parameters are duplicated for niter times.'''
def write_input(params, RUNDIR, niter=1):
  fin = open("%s/QR.in"%(RUNDIR), 'w')
  fin.write("%d\n"%(len(params) * niter))
  for param in params:
    for k in range(niter):
      fin.write("%2s%6d%6d%6d%6d%6d%6d%20.13E\n"%(param[0], param[1], param[2], param[5], param[6], param[9], param[10],param[11]))
  fin.close()

''' Execute the Fortran driver using MPI spawn. Note that the inputs and parameters are passed to Frotran using environment vairables, command lines and files. '''
def execute(nproc, nthreads, RUNDIR):
  info = MPI.Info.Create()
  info.Set('env', 'OMP_NUM_THREADS=%d\n' %(nthreads))
  print('exec', "%s/pdqrdriver"%(BINDIR), 'args', "%s/"%(RUNDIR), 'nproc', nproc)
  comm = MPI.COMM_SELF.Spawn("%s/pdqrdriver"%(BINDIR), args="%s/"%(RUNDIR), maxprocs=nproc,info=info)
  comm.Disconnect()
  return

''' Read the runtime from the output file RUNDIR/QR.out which contains the runtime for the same parameters by running QR factorization for niter times. Only the minimum among the niter runtimes is returned. '''
def read_output(params, RUNDIR, niter=1):
  fout = open("%s/QR.out"%(RUNDIR), 'r')
  times = float('Inf')
  for line in fout.readlines():
    words = line.split()
    if (len(words) > 0 and words[0] == "WALL"):
      if (words[9] == "PASSED"):
        mytime = float(words[7])
        if (mytime < times):
          times = mytime
  fout.close()
  return times

''' The Python dirver that writes parameters to individual files, runs the Fortran
     driver, and read the runtime from individual files. Note the same parameter is
     executed niter times. '''
def pdqrdriver(params, niter=10,JOBID: int = None):
  global EXPDIR  # path to the input and output files
  global BINDIR  # path to the executable
  global ROOTDIR # path to the folder "scalapack-driver"

  ROOTDIR = os.path.abspath(os.path.join(os.path.realpath(__file__), '/scalapack-
    driver'))
  BINDIR = os.path.abspath(os.path.join(ROOTDIR, "bin", MACHINE_NAME))
  EXPDIR = os.path.abspath(os.path.join(ROOTDIR, "exp", MACHINE_NAME + '/' +
    TUNER_NAME))

  if (JOBID==-1):  # -1 is the default value if jobid is not set
    JOBID = os.getpid()
  RUNDIR = os.path.abspath(os.path.join(EXPDIR, str(JOBID)))
  os.system("mkdir -p %s"%(EXPDIR))
  os.system("mkdir -p %s"%(RUNDIR))
  idxproc = 8  # the index in params representing MPI counts
  idxth = 7    # the index in params representing thread counts
  write_input(params, RUNDIR, niter=niter)
  execute(params[idxproc], params[idxth], RUNDIR)
  times = read_output(params, RUNDIR, niter=niter)
  return times

''' The objective function required by GPTune. '''
def objectives(point):
  m = point['m']
  n = point['n']
  mb = point['mb']
  nb = point['nb']
  nproc = point['nproc']
  p = point['p']

  if(nproc==0 or p==0 or nproc<p): # this become useful when the parameters
    returned by TLA1 do not respect the constraints
    print('Warning: wrong parameters for objective function!!!')
    return 1e12

  nthreads   = int(nprocmax / nproc)
  q     = int(nproc / p)
  params = [('QR', m, n, nodes, cores, mb, nb, nthreads, nproc, p, q, 1.)]
  elapsedtime = pdqrdriver(params, niter = 3)
  print(params, ' scalapack time: ', elapsedtime)
  return elapsedtime

def main():
  global nodes
  global cores
  global JOBID
  global nprocmax
  global MACHINE_NAME
  global TUNER_NAME

  mmax = 2000  # maximum row dimension
  nmax = 2000  # maximum column dimension
  ntask = 10   # 10 tasks used for MLA
  nodes = 4    # 4 nodes used for the tuner
  cores = 8    # 8 threads per node
  MACHINE_NAME = 'machinename'  # MACHINE_NAME is part of the input/output file names
  NS = 20   # 20 samples per task
  JOBID = 0    # JOBID is part of the input/output file names
  TUNER_NAME='GPTune' # TUNER_NAME is part of the input/output file names
  nprocmax = nodes*cores-1  # The maximum MPI counts for the application code. Note that 1 process is reserved as the spawning process
  nprocmin = nodes # The minimum MPI counts for the application code.

  """ Define and print the spaces and constraints """
  # Task Parameters
  m     = Integer (128 , mmax, transform="normalize", name="m")
  n     = Integer (128 , nmax, transform="normalize", name="n")
  IS = Space([m, n])
  # Tuning Parameters
  mb    = Integer (1 , 512, transform="normalize", name="mb")
  nb    = Integer (1 , 512, transform="normalize", name="nb")
  nproc = Integer (nprocmin, nprocmax, transform="normalize", name="nproc")
  p     = Integer (1 , nprocmax, transform="normalize", name="p")
  PS = Space([mb, nb, nproc, p])
  # Output
  r     = Real    (float("-Inf") , float("Inf"), name="r")
  OS = Space([r])
  # Constraints
  cst1 = "mb * p <= m"
  cst2 = "nb * nproc <= n * p"
  cst3 = "nproc >= p"
  constraints = {"cst1" : cst1, "cst2" : cst2, "cst3" : cst3}
  print(IS, PS, OS, constraints)

  problem = TuningProblem(IS, PS, OS, objective, constraints, None)
  computer = Computer(nodes = nodes, cores = cores, hosts = None)

  """ Set and validate options """
  options = Options()
  options['model_restarts'] = 4   # number of GP models being built in one iteration (only the best model is retained)
  options['distributed_memory_parallelism'] = True # True: Use MPI. One MPI per model start in the modeling phase, one MPI per task in the search phase
  options['shared_memory_parallelism'] = False # True: Use threads. One thread per model start in the modeling phase, one MPI per task in the search phase
  options.validate(computer = computer)

  """ Intialize the tuner with existing data"""
  data = Data(problem)  # intialize with empty data, but can also load data from
    previous runs
  gptune = GPTune(problem, computer = computer, data = data, options = options)

  """ Building MLA with the given list of tasks """
  giventask = [[400,500],[800,600]]
  NI = len(giventask)
  (data, models,stats) = gptune.MLA(NS=NS, NI=NI, Igiven =giventask, NS1 = max(NS
    //2,1))
  print("stats: ",stats)

  """ Dump the tuner and data to file for later use """
  pickle.dump(gptune, open('MLA_nodes_%d_cores_%d_mmax_%d_nmax_%d_machine_%
    s_jobid_%d.pkl'%(nodes,cores,mmax,nmax,machine,JOBID), 'wb'))

  """ Print all task input and parameter samples """
  for tid in range(NI):
    print("tid: %d"%(tid))
    print("    m:%d n:%d"%(data.I[tid][0], data.I[tid][1]))
    print("    Ps ", data.P[tid])
    print("    Os ", data.O[tid])
    print('    Popt ', data.P[tid][np.argmin(data.O[tid])], 'Yopt ', min(data.O[
    tid])[0])

if __name__ == "__main__":
  main()
```

Listing 2: scalapack QR example: MLA in a single run.

```
      PROGRAM PDQRDRIVER
      CHARACTER*200       FILEDIR
      INTEGER             IAM
      INTEGER             master
      PARAMETER           ( NIN = 1, NOUT = 2)
      DATA                KTESTS, KPASS, KFAIL, KSKIP /4*0/

*     Get starting information
      CALL GETARG(1,FILEDIR)
      CALL MPI_INIT(ierr)
      CALL MPI_COMM_GET_PARENT(master, ierr)
      CALL BLACS_PINFO( IAM, NPROCS )

*     Open input file
      OPEN( NIN, FILE=trim(FILEDIR)//'QR.in', STATUS='OLD' )
      IF( IAM.EQ.0 ) THEN
          OPEN( NOUT, FILE=trim(FILEDIR)//'QR.out', STATUS='UNKNOWN' )
      END IF

*     Read number of configurations
      READ( NIN, FMT = 1111 ) NBCONF
*     Print headings
      IF( IAM.EQ.0 ) THEN
          WRITE( NOUT, FMT = * )
```

```
         WRITE( NOUT, FMT = 9995 )
         WRITE( NOUT, FMT = 9994 )
         WRITE( NOUT, FMT = * )
     END IF

*     Run the computation for NBCONF times and dump the runtime to QR.out
     DO 50 CONFIG = 1, NBCONF
*       ...
*       ...
     END DO

*     Print out ending messages and close output file
     IF( IAM.EQ.0 ) THEN
        KTESTS = KPASS + KFAIL + KSKIP
        WRITE( NOUT, FMT = * )
        WRITE( NOUT, FMT = 9992 ) KTESTS
        IF( CHECK ) THEN
           WRITE( NOUT, FMT = 9991 ) KPASS
           WRITE( NOUT, FMT = 9989 ) KFAIL
        ELSE
           WRITE( NOUT, FMT = 9990 ) KPASS
        END IF
        WRITE( NOUT, FMT = 9988 ) KSKIP
        WRITE( NOUT, FMT = * )
        WRITE( NOUT, FMT = * )
        WRITE( NOUT, FMT = 9987 )
     END IF

*     Close input and output files
     CLOSE( NIN )
     IF( IAM.EQ.0 ) THEN
         CLOSE( NOUT )
     END IF

*   Disconnect the inter communicator, destroy BLACS grid and the inter communicator
    call MPI_COMM_DISCONNECT(master, ierr)
    CALL BLACS_EXIT( 1 )
    call MPI_Finalize(ierr)

*   Formats
 1111 FORMAT( I6 )
 9995 FORMAT( 'TIME      M      N  MB  NB     P     Q Fact Time ','     MFLOPS CHECK  Residual' )
 9994 FORMAT( '---- ------ ------ --- --- ----- ----- --------- ','----------- ------  ---------' )
 9992 FORMAT( 'Finished ', I6, ' tests, with the following results:' )
 9991 FORMAT( I5, ' tests completed and passed residual checks.' )
 9990 FORMAT( I5, ' tests completed without checking.' )
 9989 FORMAT( I5, ' tests completed and failed residual checks.' )
 9988 FORMAT( I5, ' tests skipped because of illegal input values.' )
 9987 FORMAT( 'END OF TESTS.' )

     STOP
```

```
      END
```

Listing 3: pdqrdriver.f.

The following example runlog illustrates how to understand the code generated information. First, the code prints out the IS, PS and OS. Then it shows the parallelization parameters being used by GPTune. Next, the code prints different phases in each MLA iteration. Next, the tuner runtime profile is printed. Finally, all and the best samples of each task are listed.

```

IS: Space([Integer(low=128, high=2000),
    Integer(low=128, high=2000)])
PS: Space([Integer(low=1, high=128),
    Integer(low=1, high=128),
    Integer(low=4, high=31),
    Integer(low=1, high=31)])
OS: Space([Real(low=-inf, high=inf, prior='uniform', transform='identity')])
constraints: {'cst1': 'mb * p <= m', 'cst2': 'nb * nproc <= n * p', 'cst3': 'nproc
    >= p'}


------Validating the options
    total core counts provided to GPTune: 32
     ---> distributed_memory_parallelism: True
     ---> shared_memory_parallelism: False
     ---> objective_evaluation_parallelism: False

    total core counts for modeling: 29
     ---> model_processes: 6
     ---> model_threads: 1
     ---> model_restart_processes: 4
     ---> model_restart_threads: 1

    total core counts for search: 32
     ---> search_processes: 1
     ---> search_threads: 1
     ---> search_multitask_processes: 31
     ---> search_multitask_threads: 1

    total core counts for objective function evaluation: 32
     ---> core counts in a single application run: 31
     ---> objective_multisample_processes: 1
     ---> objective_multisample_threads: 1


------Starting MLA with 2 tasks
MLA initial sampling:
...
MLA iteration:  0
...
MLA iteration:  9
...
stats:  {'time_total': 61.6, 'time_fun': 18.4, 'time_search': 11.9, 'time_model':
    31.1}

tid: 0
    m:400 n:500

    Ps  [[28, 47, 15, 9], [9, 124, 26, 17], [38, 50, 15, 10], [1, 117, 28, 18],
    [77, 73, 9, 3],[31, 50, 16, 7], [8, 117, 28, 19], [71, 67, 8, 2],
    [35, 39, 15, 9], [13, 116, 27, 16],[121, 102, 7, 2], [101, 128, 5, 2],
    [21, 13, 27, 3], [15, 127, 6, 2], [16, 14, 30, 2],[49, 30, 27, 2],
    [106, 6, 29, 2], [108, 31, 6, 2], [32, 14, 31, 2], [86, 5, 25, 2]]

    Os  [[0.0092][0.0157][0.01][0.0152][0.0083][0.0082][0.0171][0.0079]
        [0.0104][0.0143][0.009][0.0112][0.0068][0.0102][0.006][0.0057]
        [0.0066][0.0061][0.0059][0.0062]]

    Popt  [49, 30, 27, 2] Oopt  0.005727

tid: 1
    m:800 n:600

    Ps  [[32, 40, 16, 8], [13, 116, 28, 17], [76, 65, 7, 2], [116, 5, 30, 6],
    [33, 51, 15, 9], [5, 123, 26, 16], [65, 69, 9, 3], [116, 11, 29, 4],
    [31, 44, 17, 7], [11, 117, 28, 18],[119, 124, 26, 6], [12, 72, 21, 8],
    [87, 10, 31, 4], [1, 101, 7, 4], [122, 7, 22, 4],[40, 26, 28, 2],
    [101, 21, 26, 2], [97, 37, 12, 3], [2, 2, 29, 2], [29, 10, 30, 1]]

    Os  [[0.0136][0.0239][0.0174][0.0177][0.0141][0.0231][0.0174][0.014]
        [0.0116][0.0248][0.0237][0.015 ][0.0144][0.0169][0.0145][0.0122]
        [0.0118][0.0127][0.0201][0.0103]]

    Popt  [29, 10, 30, 1] Oopt  0.01032
```
Listing 4: runlog of MLA

5.1.2 MLA in several runs

When one does not have enough computation resource to build a complete LCM model within a single run, it becomes useful to divide the workload (number of samples per task) into multiple chunks and run the driver multiple times using a checkpoint-restart like mechanism. Each time, the driver reads the last checkpoint from a file, generates more data points if necessary and rebuild the LCM model, and then dump the new checkpoint to the file.

The following example driver, when executed the first time, will build a LCM model for two user specified tasks [[400,500],[800,600]] with "NS = 15" samples (line 18) per task, then save the sampled data to 'Data_nodes_%d_cores_%d_mmax_%d_nmax_%d_machine_%s_jobid_%d.pkl'%(nodes,cores,mmax,nmax,machine,JOBID) as a checkpoint. If we set "NS = 20" (line 18) and rerun the driver, it will add additional 5 samples per task and update the LCM model, then append the new sampled data to the same file. Note that the driver will try to load the data from file; if no previous data is available, it will create an empty data structure. See the "try ... except ..." statement at line 56.

```
1 ''' The objective function required by GPTune. '''
2 def objectives(point):
3   # ... same as the one in Section 5.1.1
4 def main():
5   global nodes
6   global cores
7   global JOBID
```

```python
global nprocmax
global MACHINE_NAME
global TUNER_NAME

mmax = 2000  # maximum row dimension
nmax = 2000  # maximum column dimension
ntask = 10   # 10 tasks used for MLA
nodes = 4    # 4 nodes used for the tuner
cores = 8    # 8 threads per node
MACHINE_NAME = 'machinename'  # MACHINE_NAME is part of the input/output file
  names
NS = 15   # 15 samples per task
JOBID = 0    # JOBID is part of the input/output file names
TUNER_NAME='GPTune' # TUNER_NAME is part of the input/output file names
nprocmax = nodes*cores-1  # The maximum MPI counts for the application code.
  Note that 1 process is reserved as the spawning process
nprocmin = nodes # The minimum MPI counts for the application code.

""" Define and print the spaces and constraints """
# Task Parameters
m     = Integer (128 , mmax, transform="normalize", name="m")
n     = Integer (128 , nmax, transform="normalize", name="n")
IS = Space([m, n])
# Tuning Parameters
mb    = Integer (1 , 128, transform="normalize", name="mb")
nb    = Integer (1 , 128, transform="normalize", name="nb")
nproc = Integer (nprocmin, nprocmax, transform="normalize", name="nproc")
p     = Integer (1 , nprocmax, transform="normalize", name="p")
PS = Space([mb, nb, nproc, p])
# Output
r     = Real    (float("-Inf") , float("Inf"), name="r")
OS = Space([r])
# Constraints
cst1 = "mb * p <= m"
cst2 = "nb * nproc <= n * p"
cst3 = "nproc >= p"
constraints = {"cst1" : cst1, "cst2" : cst2, "cst3" : cst3}
print(IS, PS, OS, constraints)

problem = TuningProblem(IS, PS, OS, objectives, constraints, None)
computer = Computer(nodes = nodes, cores = cores, hosts = None)

""" Set and validate options """
options = Options()
options['model_restarts'] = 4   # number of GP models being built in one
  iteration (only the best model is retained)
options['distributed_memory_parallelism'] = True # True: Use MPI. One MPI per
  model start in the modeling phase, one MPI per task in the search phase
options['shared_memory_parallelism'] = False # True: Use threads. One thread per
   model start in the modeling phase, one MPI per task in the search phase
options.validate(computer = computer)

""" Intialize the tuner with existing data stored as last check point"""
try:
  data = pickle.load(open('Data_nodes_%d_cores_%d_mmax_%d_nmax_%d_machine_%
  s_jobid_%d.pkl'%(nodes,cores,mmax,nmax,machine,JOBID), 'rb'))
  except (OSError, IOError) as e:
    data = Data(problem)
  gptune = GPTune(problem, computer = computer, data = data, options = options)

  """ Building MLA with NI random tasks """
  NI = ntask
  (data, models,stats) = gptune.MLA(NS=NS, NI=NI, NS1 = max(NS//2,1))
  print("stats: ",stats)

  """ Dump the data to file as a new check point """
  pickle.dump(data, open('Data_nodes_%d_cores_%d_mmax_%d_nmax_%d_machine_%s_jobid_
    %d.pkl'%(nodes,cores,mmax,nmax,machine,JOBID), 'wb'))

  """ Dump the tuner to file for TLA use """
  pickle.dump(gptune, open('MLA_nodes_%d_cores_%d_mmax_%d_nmax_%d_machine_%
    s_jobid_%d.pkl'%(nodes,cores,mmax,nmax,machine,JOBID), 'wb'))

  """ Print the optimal parameters and function evaluations"""
  for tid in range(NI):
    print("tid: %d"%(tid))
    print("    m:%d n:%d"%(data.I[tid][0], data.I[tid][1]))
    print("    Ps ", data.P[tid])
    print("    Os ", data.O[tid])
    print('    Popt ', data.P[tid][np.argmin(data.O[tid])], 'Yopt ', min(data.O[
    tid])[0])

if __name__ == "__main__":
   main()
```

Listing 5: scalapack QR example: MLA in multiple runs.

5.1.3 MLA+TLA in a single run

The following example first calls MLA to build a LCM model for the QR driver using 10 randomly generated tasks, then calls TLA to predict the optimal tuning parameters for 2 new tasks [[400,500],[800,600]].

```python
''' The objective function required by GPTune. '''
def objectives(point):
  # ... same as the one in Section 5.1.1
def main():
  global nodes
  global cores
  global JOBID
  global nprocmax
  global MACHINE_NAME
  global TUNER_NAME

  mmax = 2000  # maximum row dimension
  nmax = 2000  # maximum column dimension
  ntask = 10   # 10 tasks used for MLA
  nodes = 4    # 4 nodes used for the tuner
  cores = 8    # 8 threads per node
  MACHINE_NAME = 'machinename'  # MACHINE_NAME is part of the input/output file
    names
```

```
NS = 20   # 20 samples per task
JOBID = 0    # JOBID is part of the input/output file names
TUNER_NAME='GPTune' # TUNER_NAME is part of the input/output file names
nprocmax = nodes*cores-1  # The maximum MPI counts for the application code.
  Note that 1 process is reserved as the spawning process
nprocmin = nodes # The minimum MPI counts for the application code.

""" Define and print the spaces and constraints """
# Task Parameters
m     = Integer (128 , mmax, transform="normalize", name="m")
n     = Integer (128 , nmax, transform="normalize", name="n")
IS = Space([m, n])
# Tuning Parameters
mb    = Integer (1 , 512, transform="normalize", name="mb")
nb    = Integer (1 , 512, transform="normalize", name="nb")
nproc = Integer (nprocmin, nprocmax, transform="normalize", name="nproc")
p     = Integer (1 , nprocmax, transform="normalize", name="p")
PS = Space([mb, nb, nproc, p])
# Output
r     = Real    (float("-Inf") , float("Inf"), name="r")
OS = Space([r])
# Constraints
cst1 = "mb * p <= m"
cst2 = "nb * nproc <= n * p"
cst3 = "nproc >= p"
constraints = {"cst1" : cst1, "cst2" : cst2, "cst3" : cst3}
  print('IS:',IS)
  print('PS:',PS)
  print('OS:',OS)
  print('constraints:', constraints)

problem = TuningProblem(IS, PS, OS, objectives, constraints, None)
computer = Computer(nodes = nodes, cores = cores, hosts = None)

""" Set and validate options """
options = Options()
options['model_restarts'] = 4   # number of GP models being built in one
  iteration (only the best model is retained)
options['distributed_memory_parallelism'] = True # True: Use MPI. One MPI per
  model start in the modeling phase, one MPI per task in the search phase
options['shared_memory_parallelism'] = False # True: Use threads. One thread per
   model start in the modeling phase, one MPI per task in the search phase
options.validate(computer = computer)

""" Intialize the tuner with existing data"""
data = Data(problem)  # intialize with empty data, but can also load data from
  previous runs
gptune = GPTune(problem, computer = computer, data = data, options = options)

""" Build MLA with NI random tasks """
NI = ntask
(data, models,stats) = gptune.MLA(NS=NS, NI=NI, NS1 = max(NS//2,1))
print("stats: ",stats)

""" Dump the tuner and data to file for later use """
```

```
  pickle.dump(gptune, open('MLA_nodes_%d_cores_%d_mmax_%d_nmax_%d_machine_%
    s_jobid_%d.pkl'%(nodes,cores,mmax,nmax,machine,JOBID), 'wb'))

  """ Print all task input and parameter samples """
  for tid in range(NI):
    print("tid: %d"%(tid))
    print("    m:%d n:%d"%(data.I[tid][0], data.I[tid][1]))
    print("    Ps ", data.P[tid])
    print("    Os ", data.O[tid])
    print('    Popt ', data.P[tid][np.argmin(data.O[tid])], 'Yopt ', min(data.O[
    tid])[0])

  """ Call TLA for 2 new tasks using the constructed LCM model"""
  newtask = [[400,500],[800,600]]
  (aprxopts,objval,stats) = gptune.TLA1(newtask, NS=None)
  print("stats: ",stats)

  """ Print the optimal parameters and function evaluations"""
  for tid in range(len(newtask)):
    print("new task: %s"%(newtask[tid]))
    print('    predicted Popt: ', aprxopts[tid], ' objval: ',objval[tid])

if __name__ == "__main__":
  main()
```

Listing 6: scalapack QR example: MLA+TLA in a single run.

The following runlog illustrates how the output of TLA looks like. Please refer to Section 5.1.1 for the output of MLA.

```

IS: Space([Integer(low=128, high=2000),
    Integer(low=128, high=2000)])
PS: Space([Integer(low=1, high=128),
    Integer(low=1, high=128),
    Integer(low=4, high=31),
    Integer(low=1, high=31)])
OS: Space([Real(low=-inf, high=inf, prior='uniform', transform='identity')])
constraints: {'cst1': 'mb * p <= m', 'cst2': 'nb * nproc <= n * p', 'cst3': 'nproc
     >= p'}

------Validating the options
  total core counts provided to GPTune: 32
   ---> distributed_memory_parallelism: True
   ---> shared_memory_parallelism: False
   ---> objective_evaluation_parallelism: False

  total core counts for modeling: 3
   ---> model_processes: 1
   ---> model_threads: 1
   ---> model_restart_processes: 1
   ---> model_restart_threads: 1

  total core counts for search: 32
   ---> search_processes: 1
   ---> search_threads: 1
   ---> search_multitask_processes: 31
   ---> search_multitask_threads: 1

  total core counts for objective function evaluation: 32
   ---> core counts in a single application run: 31
   ---> objective_multisample_processes: 1
   ---> objective_multisample_threads: 1


------Starting MLA with 10 tasks
...

------Starting TLA1 for task:  [[400, 500], [800, 600]]
...
stats:  {'time_total': 1.337391381, 'time_fun': 0.906422981}
new task: [400, 500]
    predicted Popt:  [38, 15, 27, 1]  objval:  [[0.004]]
new task: [800, 600]
    predicted Popt:  [35, 15, 30, 1]  objval:  [[0.0103]]
```

Listing 7: runlog of MLA+TLA (the part of MLA is skipped)

5.1.4 TLA with loading MLA from file

This example first load a previous constructed LCM model stored in file named 'MLA_nodes_%d_cores_%d_mmax_%d_nmax_%d_machine_%s_jobid_%d.pkl'%(nodes,cores,mmax,nmax,machine,JOBID), which stores both all the data samples and the trained LCM model. Then TLA is invoked for tuning two tasks [[400,500],[800,600]].

```python
''' The objective function required by GPTune. '''
def objectives(point):
  # ... same as the one in Section 5.1.1
def main():
  global ROOTDIR
  global nodes
  global cores
  global JOBID

  mmax = 2000  # maximum row dimension
  nmax = 2000  # maximum column dimension
  nodes = 4    # 4 nodes used for the tuner
  cores = 8    # 8 threads per node
  MACHINE_NAME = 'machinename'  # MACHINE_NAME is part of the input/output file
    names
  JOBID = 0    # JOBID is part of the input/output file names
  TUNER_NAME='GPTune' # TUNER_NAME is part of the input/output file names

  os.system("mkdir -p scalapack-driver/bin/%s; cp ../build/pdqrdriver scalapack-
    driver/bin/%s/.;"%(MACHINE_NAME, MACHINE_NAME))

  """ Load the tuner and data from file """
  gptune = pickle.load(open('MLA_nodes_%d_cores_%d_mmax_%d_nmax_%d_machine_%
    s_jobid_%d.pkl'%(nodes,cores,mmax,nmax,MACHINE_NAME,JOBID), 'rb'))

  """ Call TLA for 2 new tasks using the loaded data and LCM model"""
  newtask = [[400,500],[800,600]]
  (aprxopts,objval,stats) = gptune.TLA1(newtask, 0)
  print("stats: ",stats)

  """ Print the optimal parameters and function evaluations"""
  for tid in range(len(newtask)):
    print("new task: %s"%(newtask[tid]))
    print('    predicted Popt: ', aprxopts[tid], ' objval: ',objval[tid])

if __name__ == "__main__":
   main()
```

Listing 8: scalapack QR example: TLA by loading MLA from file.

5.2 SuperLU_DIST

For a typical SuperLU_DIST driver, one can use a given sparse matrix to define a task, and consider (COLPERM, LOOKAHEAD, nproc, nprows, NSUP, NREL) to be the tuning parameters affecting the objective function (e.g., runtime). Here COLPERM is the column permutation option, LOOKAHEAD is the size of the lookahead window in the factorization, nproc is the MPI count, nprows is the number of row processes, NSUP is the maximum supernode size, and NREL is the supernode relaxation parameter. Just like the ScaLAPACK example, we use a MPI spawn approach to invoke the application code (pddrive_spawn.c). However, this example does not use files for passing information between the driver and the application. Instead, some task parameters and tuning parameters are passed to the application through command lines, while the others are passed through environment variables; the output is passed back using the spawned MPI communicator.

5.2.1 MLA+TLA in a single run

The following example first calls MLA to build a LCM model for the SuperLU_DIST driver pddrive_spawn.c using two tasks [[“g4.rua”], [“g20.rua”]], then calls TLA to predict the optimal tuning parameters for a new task [[“big.rua”]]. Note that only the simplified code is shown here, please refer to GPTune/examples/superlu_MLA_TLA.py and
GPTune/examples/superlu_dist/EXAMPLE/pddrive_spawn.c for the complete working codes.

```
def objectives(point):
  RUNDIR = os.path.abspath(__file__ + "/../superlu_dist/build/EXAMPLE") # the path
     to the executable
  INPUTDIR = os.path.abspath(__file__ + "/../superlu_dist/EXAMPLE/") # the path to
     the matrix collection

  matrix = point['matrix']
  COLPERM = point['COLPERM']
  LOOKAHEAD = point['LOOKAHEAD']
  nprows = point['nprows']
  nproc = point['nproc']
  NSUP = point['NSUP']
  NREL = point['NREL']
  nthreads   = int(nprocmax / nproc)
  npcols     = int(nproc / nprows)
  nproc     = int(nprows * npcols)
  params = [matrix, 'COLPERM', COLPERM, 'LOOKAHEAD', LOOKAHEAD, 'nthreads',
   nthreads, 'nprows', nprows, 'npcols', npcols, 'NSUP', NSUP, 'NREL', NREL]

  """ pass some parameters through environment variables """
  info = MPI.Info.Create()
  envstr= 'OMP_NUM_THREADS=%d\n' %(nthreads)
  envstr+= 'NREL=%d\n' %(NREL)
  envstr+= 'NSUP=%d\n' %(NSUP)
  info.Set('env',envstr)

  """ use MPI spawn to call the executable, and pass the other parameters and
    inputs through command line """
  comm = MPI.COMM_SELF.Spawn("%s/pddrive_spawn"%(RUNDIR), args=['-c', '%s'%(npcols
    ), '-r', '%s'%(nprows), '-l', '%s'%(LOOKAHEAD), '-p', '%s'%(COLPERM), '%s/%s'%(
    INPUTDIR,matrix)], maxprocs=nproc,info=info)

  """ gather the return value using the inter-communicator, also refer to the
    INPUTDIR/pddrive_spawn.c to see how the return value are communicated """
  tmpdata = array('f', [0,0])
  comm.Reduce(sendbuf=None, recvbuf=[tmpdata,MPI.FLOAT],op=MPI.MAX,root=mpi4py.MPI
    .ROOT)
  comm.Disconnect()
  retval = tmpdata[0]
  print(params, ' superlu time: ', retval)
  return retval

def main():
  global nprocmax
  ntask = 2   # 2 tasks used for MLA
  nodes = 2   # 2 nodes used for the tuner
  cores = 4   # 4 threads per node
  NS = 20  # 20 samples per task
  nprocmax = nodes*cores-1  # The maximum MPI counts for the application code.
    Note that 1 process is reserved as the spawning process
  nprocmin = nodes # The minimum MPI counts for the application code.
  matrices = ["big.rua", "g4.rua", "g20.rua"]


  """ Define and print the spaces and constraints """
  # Task Parameters
  matrix    = Categoricalnorm (matrices, transform="onehot", name="matrix")
  IS = Space([matrix])
  # Tuning Parameters
  COLPERM   = Categoricalnorm ([2, 4], transform="onehot", name="COLPERM")
  LOOKAHEAD = Integer     (5, 20, transform="normalize", name="LOOKAHEAD")
  nprows    = Integer     (1, nprocmax, transform="normalize", name="nprows")
  nproc     = Integer     (nprocmin, nprocmax, transform="normalize", name="nproc"
    )
  NSUP      = Integer     (30, 300, transform="normalize", name="NSUP")
  NREL      = Integer     (10, 40, transform="normalize", name="NREL")
  PS = Space([COLPERM, LOOKAHEAD, nproc, nprows, NSUP, NREL])
  # Output
  runtime   = Real        (float("-Inf") , float("Inf"), transform="normalize",
    name="r")
  OS = Space([runtime])
  # Constraints
  cst1 = "NSUP >= NREL"
  cst2 = "nproc >= nprows"
```

```python
  constraints = {"cst1" : cst1, "cst2" : cst2}
  print(IS, PS, OS, constraints)


  problem = TuningProblem(IS, PS, OS, objectives, constraints, None)
  computer = Computer(nodes = nodes, cores = cores, hosts = None)

  """ Set and validate options """
  options = Options()
  options['model_restarts'] = 4   # number of GP models being built in one iteration (only the best model is retained)
  options['distributed_memory_parallelism'] = True # True: Use MPI. One MPI per model start in the modeling phase, one MPI per task in the search phase
  options['shared_memory_parallelism'] = False # True: Use threads. One thread per model start in the modeling phase, one MPI per task in the search phase
  options.validate(computer = computer)

  """ Intialize the tuner with existing data"""
  data = Data(problem)  # intialize with empty data, but can also load data from previous runs
  gptune = GPTune(problem, computer = computer, data = data, options = options)

  """ Build MLA with the given list of tasks """
  giventask = [["g4.rua"], ["g20.rua"]]
  NI = len(giventask)
  (data, model,stats) = gptune.MLA(NS=NS, NI=NI, Igiven =giventask, NS1 = max(NS//2,1))
  print("stats: ",stats)

  """ Print all task input and parameter samples """
  for tid in range(NI):
    print("tid: %d"%(tid))
    print("    matrix:%s"%(data.I[tid][0]))
    print("    Ps ", data.P[tid])
    print("    Os ", data.O[tid])
    print('    Popt ', data.P[tid][np.argmin(data.O[tid])], 'Yopt ', min(data.O[tid])[0])

  """ Call TLA for a new task using the constructed LCM model"""
  newtask = [["big.rua"]]
  (aprxopts,objval,stats) = gptune.TLA1(newtask, NS=None)
  print("stats: ",stats)

  """ Print the optimal parameters and function evaluations"""
  for tid in range(len(newtask)):
    print("new task: %s"%(newtask[tid]))
    print('    predicted Popt: ', aprxopts[tid], ' objval: ',objval[tid])

if __name__ == "__main__":
   main()
```

Listing 9: superlu_MLA_TLA.py.

```c
int main(int argc, char *argv[])
{
    int       nprow, npcol,lookahead,colperm;
```

```
  char      **cpp, c;
  FILE *fp;
MPI_Comm parent;

  /* Intialize MPI and get the inter communicator. */
  MPI_Init( &argc, &argv );
MPI_Comm_get_parent(&parent);

  /* Read the input and parameters from command line arguments. */
for (cpp = argv+1; *cpp; ++cpp) {
  if ( **cpp == '-' ) {
    c = *(*cpp+1);
    ++cpp;
    switch (c) {
      case 'h':
      printf("Options:\n");
      printf("\t-r <int>: process rows    (default %4d)\n", nprow);
      printf("\t-c <int>: process columns (default %4d)\n", npcol);
      exit(0);
      break;
      case 'r': nprow = atoi(*cpp);     // number of row processes
        break;
      case 'c': npcol = atoi(*cpp);     // number of column processes
        break;
      case 'l': lookahead = atoi(*cpp); // size of lookahead window
        break;
      case 'p': colperm = atoi(*cpp);   // column permutation
        break;
    }
  } else { /* Last arg is considered a filename */
    if ( !(fp = fopen(*cpp, "r")) ) {          // the file storing the sparse
 matrix
        ABORT("File does not exist");
      }
    break;
  }
  }

/* Read the input and parameters from environment variables (including NSUP,
 NREL and OMP_NUM_THREADS) */
if (master process) {
  print_sp_ienv_dist(&options);
  print_options_dist(&options);
  fflush(stdout);
}

/* Allocate superlu meta-data and call the computation routine. */
//...

/* sending the results (numerical factorization time) to the parent process */
result = runtime results;
MPI_Reduce(result, MPI_BOTTOM, 1, MPI_FLOAT,MPI_MAX, 0, parent);

/* DEALLOCATE SupreLU meta-data. */
//...
```

```c
  /* Disconnect the inter communicator and finalize the intra communicator. */
  MPI_Comm_disconnect(&parent);
    MPI_Finalize();
}
```

Listing 10: pddrive_spawn.c.

5.2.2 Muti-objective MLA in a single run

The following example demonstrates the capability of multi-objective auto-tuning feature of GPTune with two objectives (runtime and memory of a sparse LU factorization). The example calls MLA to build a LCM model per objective for the SuperLU_DIST driver pddrive_spawn.c using three tasks [["big.rua"], ["g4.rua"], ["g20.rua"]]. Note that only the simplified code is shown here, please refer to GPTune/examples/superlu_MLA_MO.py and GPTune/examples/superlu_dist/EXAMPLE/pddrive_spawn.c for the complete working codes.

```python
def objectives(point):
  RUNDIR = os.path.abspath(__file__ + "/../superlu_dist/build/EXAMPLE") # the path to the executable
  INPUTDIR = os.path.abspath(__file__ + "/../superlu_dist/EXAMPLE/") # the path to the matrix collection
  matrix = point['matrix']
  COLPERM = point['COLPERM']
  LOOKAHEAD = point['LOOKAHEAD']
  nprows = point['nprows']
  nproc = point['nproc']
  NSUP = point['NSUP']
  NREL = point['NREL']
  nthreads   = int(nprocmax / nproc)
  npcols     = int(nproc / nprows)
  nproc     = int(nprows * npcols)
  params = [matrix, 'COLPERM', COLPERM, 'LOOKAHEAD', LOOKAHEAD, 'nthreads', nthreads, 'nprows', nprows, 'npcols', npcols, 'NSUP', NSUP, 'NREL', NREL]


  """ pass some parameters through environment variables """
  info = MPI.Info.Create()
  envstr= 'OMP_NUM_THREADS=%d\n' %(nthreads)
  envstr+= 'NREL=%d\n' %(NREL)
  envstr+= 'NSUP=%d\n' %(NSUP)
  info.Set('env',envstr)

  """ use MPI spawn to call the executable, and pass the other parameters and inputs through command line """
  comm = MPI.COMM_SELF.Spawn("%s/pddrive_spawn"%(RUNDIR), args=['-c', '%s'%(npcols), '-r', '%s'%(nprows), '-l', '%s'%(LOOKAHEAD), '-p', '%s'%(COLPERM), '%s/%s'%(INPUTDIR,matrix)], maxprocs=nproc,info=info)

  """ gather the return value using the inter-communicator, also refer to the INPUTDIR/pddrive_spawn.c to see how the return value are communicated """

  tmpdata = array('f', [0,0])
  comm.Reduce(sendbuf=None, recvbuf=[tmpdata,MPI.FLOAT],op=MPI.MAX,root=mpi4py.MPI.ROOT)
  comm.Disconnect()

  print(params, ' superlu time: ', tmpdata[0], ' memory: ', tmpdata[1])
  return tmpdata



def main():
  global nprocmin
  ntask = 3   # 3 tasks used for MLA
  nodes = 2   # 2 nodes used for the tuner
  cores = 4   # 4 threads per node
  NS = 20  # 20 samples per task
  nprocmax = nodes*cores-1  # The maximum MPI counts for the application code. Note that 1 process is reserved as the spawning process
  nprocmin = nodes # The minimum MPI counts for the application code.
  matrices = ["big.rua", "g4.rua", "g20.rua"]

  """ Define and print the spaces and constraints """
  # Task Parameters
  matrix    = Categoricalnorm (matrices, transform="onehot", name="matrix")
  IS = Space([matrix])
  # Tuning Parameters
  COLPERM   = Categoricalnorm ([2, 4], transform="onehot", name="COLPERM")
  LOOKAHEAD = Integer     (5, 20, transform="normalize", name="LOOKAHEAD")
  nprows    = Integer     (1, nprocmax, transform="normalize", name="nprows")
  nproc     = Integer     (nprocmin, nprocmax, transform="normalize", name="nproc")
  NSUP      = Integer     (30, 300, transform="normalize", name="NSUP")
  NREL      = Integer     (10, 40, transform="normalize", name="NREL")
  PS = Space([COLPERM, LOOKAHEAD, nproc, nprows, NSUP, NREL])
  # Output
  runtime   = Real        (float("-Inf") , float("Inf"), transform="normalize", name="r")
  memory    = Real        (float("-Inf") , float("Inf"), transform="normalize", name="memory")
  OS = Space([runtime, memory])
  # Constraints
  cst1 = "NSUP >= NREL"
  cst2 = "nproc >= nprows"
  constraints = {"cst1" : cst1, "cst2" : cst2}
  print(IS, PS, OS, constraints)
  problem = TuningProblem(IS, PS, OS, objectives, constraints, None)
  computer = Computer(nodes = nodes, cores = cores, hosts = None)

  """ Set and validate options """
  options = Options()
  # options['model_processes'] = 1
  # options['model_threads'] = 1
  options['model_restarts'] = 1
  # options['search_multitask_processes'] = 1
  # options['model_restart_processes'] = 1
  options['distributed_memory_parallelism'] = False
  options['shared_memory_parallelism'] = False
  options['model_class '] = 'Model_LCM'
  options['verbose'] = False
  options['search_algo'] = 'nsga2' #'maco' #'moead' #'nsga2' #'nspso'
  options['search_pop_size'] = 1000
  options['search_gen'] = 10
  options['search_best_N'] = 4
  options.validate(computer = computer)

  """ Set and validate options """
  options = Options()
  options['model_restarts'] = 1   # number of GP models being built in one
    iteration (only the best model is retained)
  options['distributed_memory_parallelism'] = False # True: Use MPI. One MPI per
    model start in the modeling phase, one MPI per task in the search phase
  options['shared_memory_parallelism'] = False # True: Use threads. One thread per
     model start in the modeling phase, one MPI per task in the search phase
  options['search_algo'] = 'nsga2' # multi-objective search algorithm
  options['search_pop_size'] = 1000 # Population size in pgymo
  options['search_gen'] = 10 # Number of evolution generations in pgymo
  options['search_best_N'] = 4 # Maximum number of points selected using a multi-
    objective search
  options.validate(computer = computer)

  """ Intialize the tuner with existing data"""
  data = Data(problem) # intialize with empty data, but can also load data from
    previous runs
  gptune = GPTune(problem, computer = computer, data = data, options = options)

  """ Building MLA with the given list of tasks """
  giventask = [["big.rua"], ["g4.rua"], ["g20.rua"]]
  NI = len(giventask)
  (data, models,stats) = gptune.MLA(NS=NS, NI=NI, Igiven =giventask, NS1 = max(NS
    //2,1))
  print("stats: ",stats)

  """ Print all task input and parameter samples; search for and print the Pareto
    front"""
  for tid in range(NI):
    print("tid: %d"%(tid))
    print("    matrix:%s"%(data.I[tid][0]))
    print("    Ps ", data.P[tid])
    print("    Os ", data.O[tid])
    ndf, dl, dc, ndr = pg.fast_non_dominated_sorting(data.O[tid])
    front = ndf[0]
    # print('front id: ',front)
    fopts = data.O[tid][front]
    xopts = [data.P[tid][i] for i in front]
    print('    Popts ', xopts)
    print('    Oopts ', fopts)

if __name__ == "__main__":
  main()
```

Listing 11: superlu_MLA_MO.py.

6 Numerical experiments

6.1 Parallel speedups of GPTune

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

$$y_{demo}(t,x) = \exp\left(-(x+1)^{t+1}\right)\cos(2\pi x)\sum_{i=1}^{3}\sin\left(2\pi x(t+2)^i\right) \tag{7}$$

where t and x denote the task and tuning parameters. Note that this function is highly non-convex and we are interested in finding the minimum for $x \in [0,1]$ for multiple tasks t. Fig. 2 plots $y_{demo}(t,x)$ versus x for four different values of t and marks the minimum objective function values.

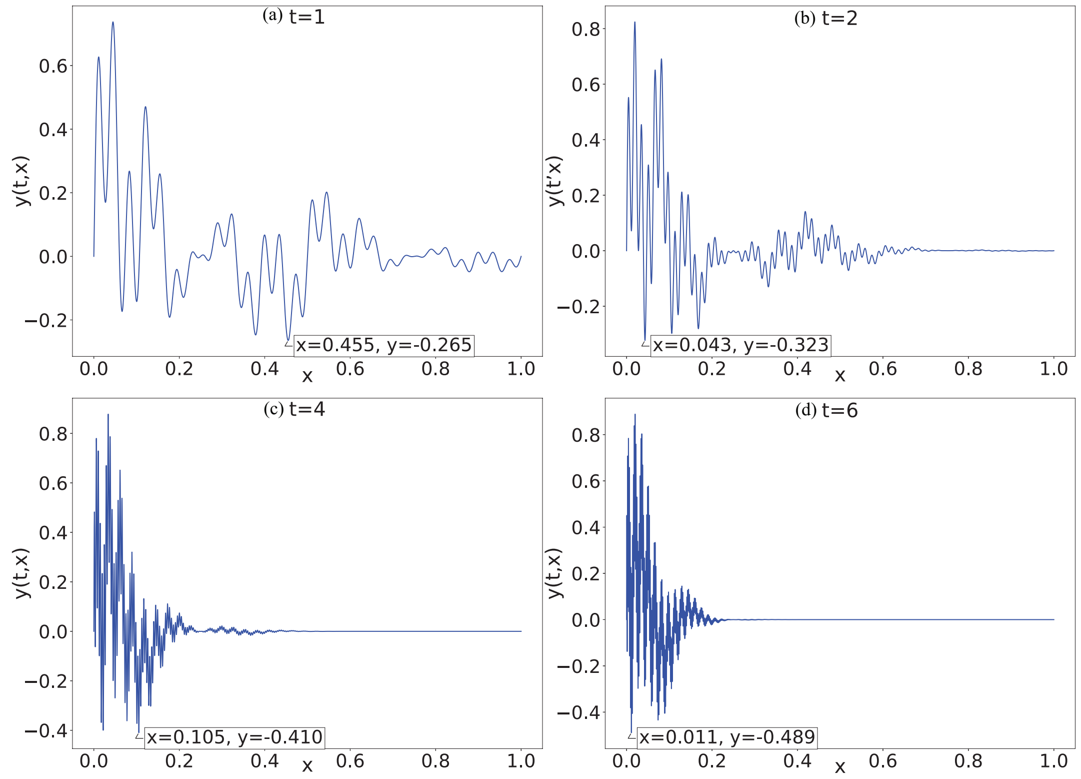

Figure 2: The objective functions in (7) for four task parameter values t.

First, we evaluate the parallel performance of the MLA algorithm on a Threadripper 1950X 16-core processor using $\delta = 20$ tasks. In Fig. 3, we plot the runtime of the modeling and search phases using 1 and 16 cores, by enabling the GPTune parameter distributed_memory_parallelism. For simplicity, we set the initial random sample count to NS1 = NS-1 (i.e., only one MLA iteration is performed). As we increase the number of total samples NS from 10 to 160 (with the LCM kernel matrix size changing from 200 to 3200), 13X (comparing the two blue curves) and 10X (comparing the two black curves) speedups are observed for the modeling and sampling phases, respectively.

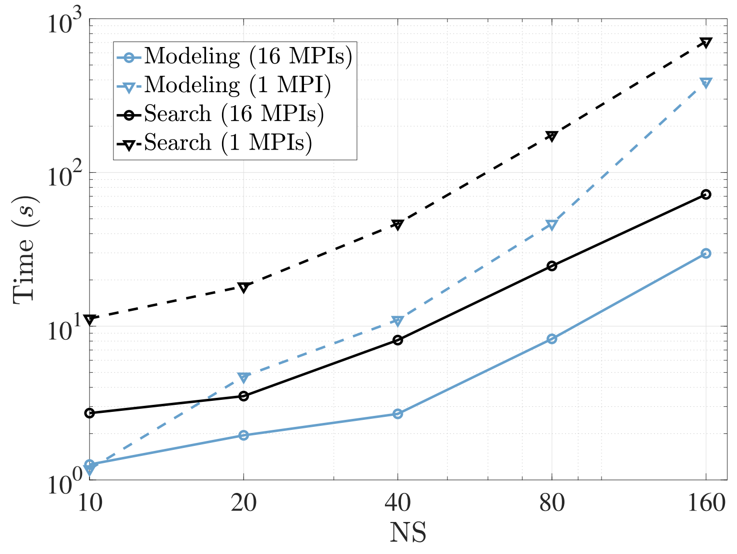

Figure 3: Modeling and search time for 1 and 16 MPIs using the objective function f_{demo}.

6.2 Advantage of using performance models

Next, we evaluate the effects of the performance models using the above objective function $y_{demo}(t, x)$. We test three performance models seperately, $\tilde{y}_1(t, x) = y_{demo}(t, x)$ (the model is excatly the objective), $\tilde{y}_2(t, x) = 10y_{demo}(t, x)$ (the model output is a factor of 10 larger than the objective), and $\tilde{y}_3(t, x) = (1 + 0.1 \times r(x))y_{demo}(t, x)$ (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 set NS1 = NS/2 and use a single task $t = 6$. Table 2 lists the minimum objective value returned by GPTune's MLA algorithm with varying total sample counts NS. Without any performance model, MLA has still not found the minimum after 640 samples. With the exact model $\tilde{y}_1$ (which is not practical), not surprisingly, at most 20 samples are sufficient to get very close to the minimum: -4.89E-01 (see Fig. 2(d)). With the scaled models $\tilde{y}_2$ and $\tilde{y}_3$, at most 80 samples and 40 samples are sufficient, respectively. In other words, with the coarse performance models, GPTune requires significantly fewer samples to build an accurate LCM model.

NS	10	20	40	80	160	320	640
None	-1.40E-01	-6.06E-02	-2.93E-02	-3.79E-01	-2.69E-01	-4.25E-01	-3.83E-01
$\tilde{y}_1$	-4.51E-01	-4.88E-01	-4.85E-01	-4.88E-01	-4.86E-01	-4.89E-01	-4.89E-01
$\tilde{y}_2$	-2.98E-01	-3.72E-01	-4.83E-01	-4.89E-01	-4.89E-01	-4.88E-01	-4.89E-01
$\tilde{y}_3$	-4.52E-01	-4.52E-01	4.88E-01	-4.77E-01	-4.89E-01	-4.89E-01	-4.89E-01

Table 2: Minimum found by GPTune for the objective f_{demo} with and without performance models.

	total time	objective evaluation	modeling	search
Single-task	15092.4	14062.3	907.8	120.1
Multi-task	9386.8	9091.4	85.7	208.1

Table 3: Runtime of different phases in the GPTune single-objective and multi-objective MLA with a total of 400 samples.

6.3 Efficiency of multi-task learning

Next, we use the ScaLAPACK QR example in Section 5.1.1 to compare the performance of the GPTune MLA algorithms with single-task (δ=1) and multi-task (δ=20) settings. We use 16 NERSC Cori nodes assuming a fixed budget of $\delta \times \text{NS} = 400$ and $\text{NS1} = \text{NS}/2$. For δ=1, we consider the task ($m = 4674, n = 3608$); for δ=20, we also consider 19 other tasks that are randomly generated with $m, n < 5000$ (in practice, one may choose all 20 tasks of interest).

Table 3 shows the runtime breakdown of the single-task and multi-task MLA algorithms. For this example, the total runtime is dominated by the objective function evaluation. The multi-task MLA requires less objective evaluation time as it involves 19 other less expensive tasks. In addition, the multi-task modeling phase is much faster than single-task one as it requires fewer MLA iterations. More specifically, the multi-task modeling requires 10 iterations with the LCM matrix dimensions $200, 220, 240, ..., 380$ while the single-task modeling requires 200 iterations with the LCM matrix dimensions $200, 201, 202, ..., 399$.

Fig. 4 plots the runtime (obtained through runing the application) and corresponding GFlops using the optimal tuning parameters for all 20 tasks. The red dots correspond to δ=20 and the blue dots correspond to δ=1. Note that the surfaces are constructed via the Matlab "griddata" function using the red dots. The multi-task MLA not only achieves a very similar minimum to the single-task MLA for ($m = 4674, n = 3608$), but also finds minima for all the other 19 tasks.

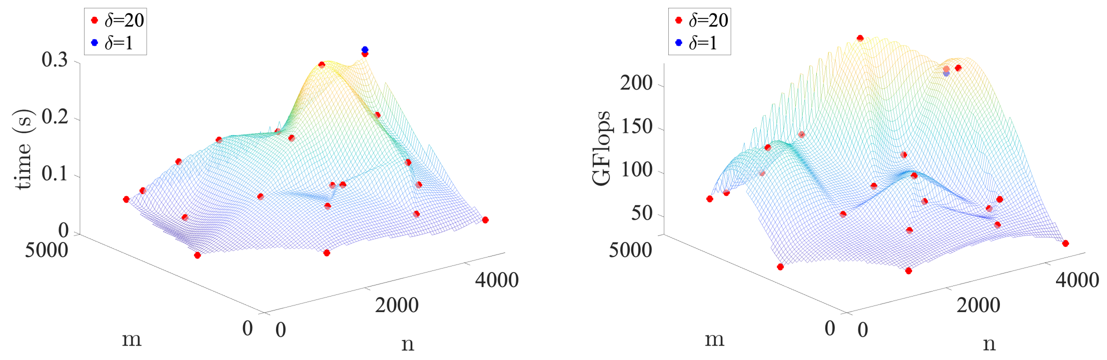

Figure 4: (a) Runtime (the objective) and (b) GFlops for 20 tasks of QR factorization after autotuning.

6.4 Capability of multi-objective tuning

Finally, we illustrate the multi-objective feature of GPTune for tuning the factorization time and memory in SuperLU_DIST [12]. As explained in Section 5.2, we consider six tuning parameters (COLPERM, LOOKAHEAD, nproc, nprows, NSUP, NREL) and two objectives (time, memory). As an example, we apply sparse factorization to a matrix “Si2” (single task) from the SuiteSparse Matrix Collection [4] using 8 NERSC Cori nodes.

As a reference, we also consider single-objective (time) and (memory). For example, single-objective (memory) tuning means minimizing the memory usage ignoring the impact on runtime (as long as the code still runs correctly). Table 4 lists the default and optimal (single-objective) tuning parameters. The default parameters are those used by SuperLU_DIST without any tuning. The optimal ones are vastly different from the default ones.

Fig. 5 plots the objective function values (via running the application) on the logarithmic scale using the default tuning parameters, and those returned by the GPTune single-objective and multi-objective MLA algorithms. The multi-objective MLA algorithm returns multiple tuning parameter configurations and their objective function values (in black), among which no data point dominates over any other in both objectives. 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.

	COLPERM	LOOKAHEAD	nproc	nprows	NSUP	NREL
Default	4	10	256	16	128	20
Single-objective (time)	2	6	216	149	295	37
Single-objective (memory)	2	5	193	20	31	22

Table 4: Default tuning parameters and optimal ones returned by the GPTune single-objective MLA algorithm.

References

[1] mpi4py. `https://pypi.org/project/mpi4py/`.

[2] PyGMO. `https://esa.github.io/pygmo/`.

[3] Scikit-Optimize. `https://scikit-optimize.github.io`.

[4] Suitesparse matrix collection. `https://sparse.tamu.edu/`.

[5] 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, Philadelphia, PA, 1997.

[6] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan. A fast and elitist multiobjective genetic algorithm: Nsga-ii. *IEEE transactions on evolutionary computation*, 6(2):182–197, 2002.

[7] P. Frazier. A Tutorial on Bayesian Optimization. `https://arxiv.org/abs/1807.02811`, 2018.

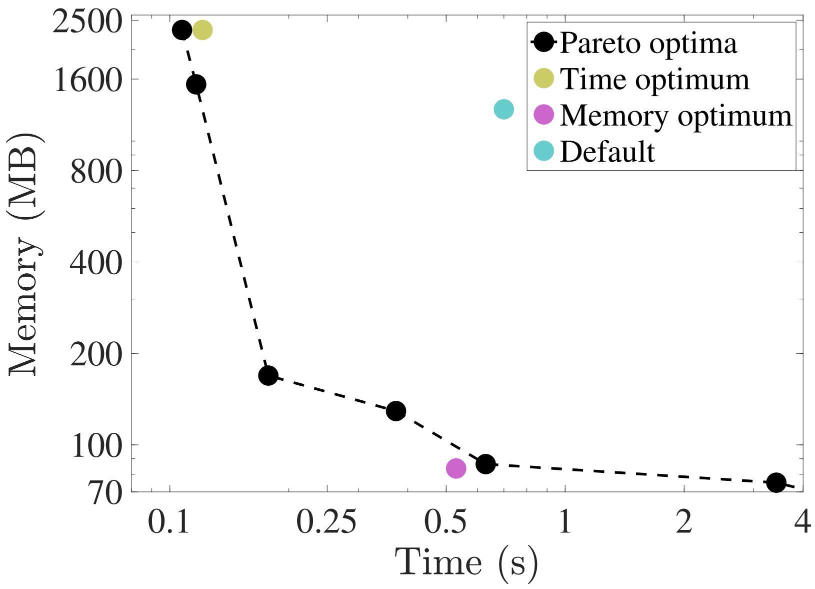

Figure 5: Logarithmic plots of the optimal objective functions values (factorization time and memory of SuperLU_DIST with 8 NERSC Cori nodes) found by GPTune using single-objective and multi-objective tuning. The objective function values using the default tuning parameters are also plotted.

[8] GPy. GPy: A gaussian process framework in python. `http://github.com/SheffieldML/GPy`, since 2012.

[9] Jason Ansel and Shoaib Kamil and Kalyan Veeramachaneni and Jonathan Ragan-Kelley and Jeffrey Bosboom and Una-May O'Reilly and Saman Amarasinghe. OpenTuner: An Extensible Framework for Program Autotuning. In *International Conference on Parallel Architectures and Compilation Techniques*, Edmonton, Canada, August 2014.

[10] D. R. Jones, M. Schonlau, and W. J. Welch. Efficient global optimization of expensive black-box functions. *Journal of Global optimization*, 13(4):455–492, 1998.

[11] J. Kennedy and R. Eberhart. Particle swarm optimization. In *Proceedings of ICNN'95 - International Conference on Neural Networks*, volume 4, pages 1942–1948 vol.4, Nov 1995.

[12] X. S. Li and J. W. Demmel. SuperLU_DIST: a scalable distributed-memory sparse direct solver for unsymmetric linear systems. *ACM Trans. Math. Softw.*, 29(2):110–140, June 2003.

[13] D. C. Liu and J. Nocedal. On the limited memory bfgs method for large scale optimization. *MATHEMATICAL PROGRAMMING*, 45:503–528, 1989.

[14] W. Sid-lakhdar, M. Aznaveh, X. Li, and J. Demmel. Multitask and Transfer Learning for Autotuning Exascale Aplications. `https://arxiv.org/abs/1908.05792`, 2019.

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