GPTune User Guide

package version: 4.0, release date: October 6, 2022

Younghyun Cho* James W. Demmel* Grace Dinh* Xiaoye S. Li†
Yang Liu† Hengrui Luo†* Osni Marques † Wissam M. Sid-Lakhdar ‡

Abstract

This user guide provides detailed instructions on how to use and understand the GPTune package, an autotuning framework based on Bayesian optimization to find the best values of performance tuning parameters for high-performance computing applications, where function evaluations are expensive and typically only a small number of function evaluations are available.

GPTune builds Gaussian Process (GP) surrogate models for Bayesian optimization with advanced features like multi-task learning, transfer learning, multi-objective tuning, multi-fidelity tuning, ensembles of multiple models, additive GP and hybrid models, sensitivity analysis, and a shared historical database. While many of these features are already published by the GPTune team in scientific journals and conference proceedings, there are some not yet published features documented in this user guide.

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.

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

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

‡Innovative Computing Laboratory, University of Tennessee, Knoxville, TN 37996. (wissam@icl.utk.edu).

Contents

1 Introduction

GPTune is an autotuning framework to solve the underlying black-box optimization problem, based on surrogate modeling such as Gaussian Process (GP) regression with support for novel autotuning features such as multi-task learning, transfer learning, multi-objective tuning, multi-fidelity tuning, incorporating cheap performance models, additive GP and hybrid models, sensitivity analysis, and a shared historical database. GPTune is particularly designed to tune High Performance Computing (HPC) applications that can contain many tuning parameters and require many computational resources to evaluate. GPTune has been applied to tune various HPC applications of the Exascale Computing Project (ECP). GPTune can also be used to tune any type of black-box tuning problem and has been applied to tune applications in various domains including scientific computing and machine learning.

1.1 Features

The goal of autotuning is to automatically choose tuning parameters to optimize the performance of an application without much human intervention. Performance of the application 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 parameters is large, the performance is a complicated and hard-to-model function of all the tuning parameters, and running and measuring the actual performance of the application is expensive. GPTune addresses these autotuning challenges by using Bayesian optimization with many advanced autotuning features, as outlined below.

Bayesian optimization. GPTune builds a GP surrogate model for Bayesian optimization by running the application and measuring its performance at a few carefully chosen tuning parameter values and optimizes the application by choosing the tuning parameters to minimize the runtime predicted by the model. To simplify the 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 algorithms choices). GPTune's interface accommodates real, integer, and categorical-type parameters.

Multi-task learning. GPTune goes beyond this simple case by providing Multi-task Learning-based Autotuning (MLA). MLA means using performance data from multiple tasks (e.g., 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 where 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 thus 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 of the values $t = t_i$ and $x = x_{opt}$, and these data are 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.

Transfer learning. GPTune also provides Transfer Learning-based Autotuning (TLA) which goes beyond MLA by leveraging knowledge of historical tuning data to tune a new task for which no actual performance data have been collected. GPTune offers two types of TLA approaches.

The first type allows GPTune to collect (extra) actual performance data for a new task, to update (rather than rebuild) the surrogate model together with the knowledge of a pre-trained MLA model, and to predict optimal x for the new tasks. GPTune provides multiple transfer learning methods to combine the knowledge of historical performance data of source tasks with the new task. The second type, on the other hand, builds a surrogate model to predict the performance of a new task without using actual performance data from the new task. The simplest way to do this is to use the same GP 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.

History database and crowd-tuning. It is often the case that function evaluation data will be collected over time by multiple users running many different instances of a popular application, so we want to take advantage of all this data to improve the accuracy of a surrogate model while reducing the cost of expensive function evaluations. GPTune History Database [5] allows historical performance data to be stored in files and the shared repository, and reused later. We provide a shared public database at `https://gptune.lbl.gov`, where users can store their performance data or download the performance data provided by other users. For reliability and security, we allow only registered users to upload and share performance data via the shared database (users can register at `https://gptune.lbl.gov/account/signup/`) . The user can specify a certain level of accessibility of each performance data sample, e.g., publicly available, private, or shared with registered users or specific user groups. Furthermore, we provide a useful Python Application Programming Interface (API), called crowd-tuning API that allows users to easily access the shared repository via Python and query relevant performance data (e.g., function evaluations or pre-trained surrogate models). The interface allows users to improve the quality of autotuning through TLA and sensitivity analysis, which we call *crowd-based autotuning* or *crowd-tuning* .

Constraint and error handling. GPTune accommodates constraints on the tuning parameters. For example, the user may require that the number of processors used (a tuning parameter) is 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" is less than the same upper bound. GPTune allows users to supply certain constraints using a user-defined function, where users can define a certain constraint through exact equalities and inequalities. GPTune also supports a Reverse Communication Interface (RCI) for handling runtime failures.

Unified support for multiple models. GPTune provides a unified interface that allows the user to invoke different autotuners. So far, other existing tuners, such as OpenTuner [21], HpBandSter [11] and our own variants of GPTune, such as GPTuneBand [41], cGP [28], and hybrid (GPTuneHybrid) models [27] are supported. This feature allows users to easily call the best autotuner for their own purposes. For example, GPTuneBand [41] handles multi-fidelity tuning problems by considering both cheap low-fidelity and expensive high-fidelity versions of objective functions with advanced kernel constructions and genetic search. cGP [28] detects the change-of-regime and non-smoothness of the objective function. GPTuneHybrid [27] efficiently explores a large discrete and categorical space, for example, a list of different algorithms, using Monte Carlo Tree Search (MCTS) techniques, and continuous space using regular GP, using dynamically selected GP surrogates.

Incorporation of cheap performance models. It is sometimes the case that a user has a possibly cheap 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

	Tuner	MLA	TLA	multi-fidelity	multi-objective	parallel tuning	database logging	checkpoint & restart
	GPTune	✓	✓		✓	✓	✓	✓
GPTune variants	GPTuneBand	✓		✓		✓	✓	✓
	cGP						✓	
	GPTuneHybrid						✓	
External	HpBandSter						✓	
	OpenTuner						✓	

Table 1: Status of the advanced features for GPTune, its variants, and external tuners it interfaces with. ✓denotes a currently supported feature.

model GPTune builds, and can help accelerate tuning.

Multi-objective tuning. If the measured performance data consist of multiple quantities, such as runtime and accuracy, then the user may want to perform multi-objective optimization. For example, in the case of runtime and accuracy, which are likely to trade off against one another (faster runtime leading to worse accuracy), then the user may want to compute the Pareto front of these two quantities. GPTune supports multi-objective tuning using a multi-objective evolutionary algorithm on the surrogates to achieve the Pareto front of the objectives, as well as support for specifying an output constraint for each of the objectives.

Scalable parallel tuning. GPTune is designed for efficient tuning for small- and large-scale applications. We provide multi-level parallelism in the framework for accelerating tuning. We have parallelized the most time-consuming part of the GPTune algorithms. Section 3.2 describes the parallel programming model on multicore nodes using Message Passing Interface (MPI) and OpenMP . GPTune also provides a serial and lite mode (Section 2.1), where the user can install and use GPTune without requiring distributed memory-level parallelism for GPTune's internal computation.

Table 1 summarizes the status of advanced features in GPTune and its variants, as well as external tuners that are interfaced with GPTune.

1.2 Notation

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; in general, multiple task parameters are 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 that is being optimized, such as time.

Table 2 summarizes the notation used in this user guide. As an illustrative example, the QR factorization routine of $ScaLAPACK$ [3], denoted as $PDGEQRF$, is used as an application code to be tuned assuming fixed numbers of compute nodes ($nodes$) and cores per node ($cores$). 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 OpenMP threads per MPI process $nthreads$ (used in BLAS), and number of column processes q, can be calculated as $nthreads = \lfloor cores/npernode \rfloor$ and $q = \lfloor nodes * npernode/p \rfloor$ (for one MPI process, by default one thread is

mapped to one core). Note that we can use modified parameter definitions to better enforce certain constraints. For example, assuming that *cores* is a power of 2, we can use a modified parameter $lg2npernode$ instead of $npernode$ such that $npernode = 2^{lg2npernode}$.

	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, message count, etc.)
	α	dimension of $\mathbb{IS}$
	β	dimension of $\mathbb{PS}$
	γ	dimension of $\mathbb{OS}$
	$\tilde{\gamma}$	dimension of $\mathbb{MS}$
	n_i (NI)	number of tasks
	n_s (NS, NS1)	number of samples per task
	$T \in \mathbb{IS}^{n_i}$	array of tasks selected from sampling
	$X \in \mathbb{PS}^{n_i \times n_s}$	array of samples (parameter configurations)
	$Y \in \mathbb{OS}^{n_i \times n_s}$	array of output results (e.g., runtime)
	Example: parameters for ScaLAPACK *PDGEQRF* notations	
Task	m	number of matrix rows
Task	n	number of matrix columns
Tuning	mb	row block size
Tuning	nb	column block size
Tuning	$npernode$	number of MPI processes per compute node
Tuning	p	number of row processes

Table 2: Notations. The symbols in the parentheses denote code notations.

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

- Parameters associated with the application codes, as described above, i.e., task parameters (predetermined as input to GPTune) and tuning parameters (tunable and optimizable by GPTune). Section 6.1.6 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 (Section 6.2.4) and TLA (6.2.5 and 6.2.6. Section 6.1.10 shows the list of all available GPTune parameters (GPTune options). The settings of these parameters can affect the speed and accuracy of the tuning algorithms.

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

2 Installation

GPTune is implemented in Python and C, and it depends on several Python, Fortran and C packages. GPTune has been used for various user systems, ranging from small-scale laptops to large-scale supercomputers. To best support the various needs of different users, we provide several GPTune installation modes, as outlined below. Users may choose an installation mode depending on their needs, and follow the detailed description in the corresponding subsection.

- Lite version of GPTune (see Section 2.1 for its manual installation): This version (also called the lite mode) installs the minimal functionality of GPTune, with fewer dependent software packages compared to the full GPTune mode. We recommend using this mode if the user has difficulties in setting up some dependent software packages in the full version or does not want to install any MPI-related software. To install this mode, **Python** with minimum version number 3.7 is needed.

- Full version of GPTune (see Section 2.2 for its manual installation): This version (also called the full mode) installs full functionality of GPTune which supports distributed parallelism for surrogate modeling and provides all the available GPTune options. Before installation, the following software environment is needed (with the minimum version number required in parentheses): **gcc** (7.4.0) or **Intel icc** (19.0.0), **OpenMPI** (4.0.1), **Python** (3.7), **ScaLAPACK** (2.1.0), **git**, **CMake** (3.19), **jq** (1.6), **BLAS** and **LAPACK** (3.4.0, but we recommend using the latest version).

- Prepared installation scripts (Section 2.3): This section introduces several prepared installation scripts that would work with full functionality for popular user systems (e.g., Mac/Linux-based laptop) and several DOE machines (e.g., NERSC's Cori/Perlmutter). If the user is using this type of known machine, we strongly recommend using the example build script for the given system and modifying the script if necessary.

- Install GPTune using the Nix build system (Section 2.4: This mode installs the full version of GPTune, and all its dependencies, using the Nix build system/package manager. Note that Nix installs self-contained copies of all of GPTune's dependencies. As a result, a Nix-based GPTune install will neither affect nor be affected by existing packages installed on the system; as a result, Nix will not be able to use preinstalled system packages (e.g. ones patched for or granted special privileges on a specific system). This is not an issue for most personal computers and workstations, but more controlled environments that require higher security levels (e.g., DOE HPC centers) may not work with the default Nix installation.

- Install GPTune using Spack (Section 2.5): This mode installs the full version of GPTune using Spack. Therefore, it is not necessary to manually install GPTune's dependencies. Note that installing all dependencies from scratch can take more than 10 hours.

- Using a pre-built Docker image (Section 2.6): A pre-built Docker image is also available, which contains the full version of GPTune. It does not require users to set up or install any dependent software packages, as long as they have already set up a Docker environment. We recommend this mode if the user wants to test GPTune in a quick way, or if the user allows running their application in a Docker environment. However, we do not recommend using this Docker image for production-level tuning due to the limitations of Docker. For example,

it is inconvenient to access files within a Docker container and Docker may not fully support MPI runtime system features.

A few known installation issues (frequently asked questions) can be found in Section 7. In the following subsections, we explain in detail each installation method. An intuitive guideline regarding which installation mode to use can be found in Fig. 1.

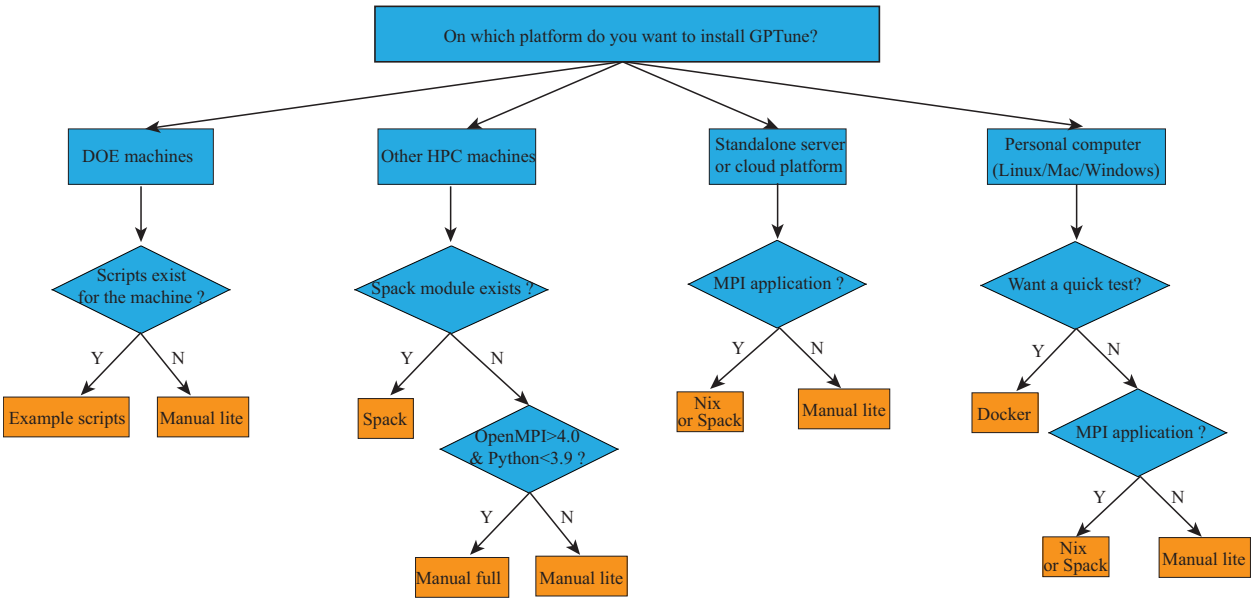

Figure 1: Guideline for the installation mode for different types of machines. "Manual lite": manual installation of the lite mode, see Section 2.1, "Manual full": manual installation of the full mode, see Section 2.2, "Example scripts": installation of the full mode using example scripts, see Section 2.3, "Nix": installation of the full mode using the Nix installer, see Section 2.4, "Spack": installation of the full mode using Spack, see Section 2.5, and "Docker": loading the pre-built Docker image of the full mode, see Section 2.6.

2.1 Lite mode

Given that the full version of GPTune can have many software dependencies (including C/Fortran and MPI-based packages), which can be time-consuming and error-prone to install everything from scratch (see Sections 2.2 or 2.3 for instructions for full installation), we provide a lite GPTune version , which relies on many fewer dependencies. The lite version only relies on **Python** (3.7).

The lite version can be useful for users who do not wish/need to install any MPI software on their computer system, e.g., those who are using a single node computer and tuning only shared-memory codes. Note that one can still use the lite version for an MPI-compiled application, please refer to Section 3.2.5 for a working example with more details.

Compared to the full version, the missing capabilities of the lite version are as follows:

- No distributed-parallel autotuning. The lite version does not use any MPI-related features and does not support the distributed-parallel surrogate modeling technique.

- Fewer options to choose a Python package for initial sampling and parameter search. The full version supports using OpenTURNs and LHSMDU for initial sampling, and PyGMO, pymoo and SciPy for tuning parameter search. The lite version does not support OpenTURNs for initial sampling and PyGMO for parameter search, which can possibly reduce the quality of the tuning results.

To start the installation, first obtain GPTune from GitHub:

```
git clone https://github.com/gptune/GPTune.git
cd GPTune
export GPTUNEROOT=$PWD
```

$PWD should indicate the current path in the user's Bash/Shell. Most of the Python dependencies can be installed using the requirements_lite.txt file, while the scikit-optimize and autotune packages need to be built from source, because GPTune applies some patches to the scikit-optimize and autotune packages for GPTune functionalities.

2.1.1 Python packages in the requirement file

The following Python packages listed in requirement_lite.txt will be installed automatically with pip: **wheel**, **numpy**, **joblib**, **scikit-learn**, **scipy**, **statsmodels**, **pyaml**, **matplotlib**, **GPy**, **lhsmdu**, **ipyparallel**, **opentuner**, **hpbandster**, **filelock**, **requests**, **pymoo**, **cloudpickle**, **SALib**, **scikit-optimize**, **cGP**, **GPTuneHybrid**, and **autotune**.

```
pip install --user -r requirements_lite.txt
```

2.1.2 Test the installation (lite version)

The basic functionality of GPTune can be tested using a synthetic function, in examples/GPTune-Demo.

Set the tuning meta description. First, modify the machine name, processor architecture, number of compute nodes, number of cores per node, and software dependencies in the meta json file at

```
$GPTUNEROOT/examples/GPTune-Demo/.gptune/meta.json
```

Invoke the tuning experiments. One can invoke the basic tuning example via

```
export GPTUNE_LITE_MODE=1
cd $GPTUNEROOT/examples/GPTune-Demo
python ./demo.py -nrun 20 # 20 is the total number of function evaluations
```

Note that the variable GPTUNE_LITE_MODE disables GPTune features that are only available in the full version of GPTune.

2.2 Full mode

The following installs the full version of GPTune on a Linux system.

First, obtain GPTune from GitHub.

```
git clone https://github.com/gptune/GPTune.git
cd GPTune
export GPTUNEROOT=$PWD
```

Set the following environment variables to make it convenient to install the GPTune dependencies. Note: see 2.2.4, 2.2.2, 2.2.3 and 2.2.5 for installing OpenMPI, CMake, BLAS/LAPACK and ScaLAPACK from source, before setting the following environment variables.

```
export PYTHONWARNINGS=ignore
export MPICC=path-to-the-mpicc-wrapper
export MPICXX=path-to-the-mpic++-wrapper
export MPIF90=path-to-the-mpif90-wrapper
export MPIRUN=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
export SITE_PACKAGES_PATH=path-to-your-site-packages # one can get the path of
    site-packages by ``pip list -v"
```

2.2.1 jq

The GPTune database is based on a JSON file format, and jq is a lightweight and flexible command-line JSON processor `https://github.com/stedolan/jq`. Most UNIX-like systems have jq available. For example, MacOS has it via homebrew; Ubuntu/Debian-like systems have it via apt-get. Alternatively, one can try to install it from source:

```
cd $GPTUNEROOT
wget https://github.com/stedolan/jq/releases/download/jq-1.6/jq-1.6.tar.gz
tar -xvf jq-1.6.tar.gz
cd jq-1.6
autoreconf -i
./configure --disable-maintainer-mode
make -j16
export PATH=$GPTUNEROOT/jq-1.6/:$PATH
```

2.2.2 CMake

CMake `https://cmake.org/` is an open-source platform for building software pipelines, which we use to build GPTune's C codes. Most systems have CMake available. But if one does not have CMake$\geq$3.19 installed, use the following to install it from source. It may take about 1 hour.

```
cd $GPTUNEROOT
version=3.19
build=1
wget https://cmake.org/files/v$version/cmake-$version.$build.tar.gz
tar -xzvf cmake-$version.$build.tar.gz
cd cmake-$version.$build/
./bootstrap
make -j4
make install
export PATH=$GPTUNEROOT/cmake-$version.$build/bin/:$PATH
```

2.2.3 BLAS/LAPACK

The BLAS `https://www.netlib.org/blas` and LAPACK `https://www.netlib.org/lapack` are the standard modern numerical libraries for basic linear algebra operations. Most systems have BLAS and LAPACK installed. For example, HPC systems typically have those modules available; MacOS has them via homebrew; Ubuntu/Debian-like systems have them via apt-get. Alternatively, one can try to install them from source, e.g., using OpenBLAS:

```
cd $GPTUNEROOT
git clone https://github.com/xianyi/OpenBLAS
cd OpenBLAS
make PREFIX=. CC=$MPICC CXX=$MPICXX FC=$MPIF90 -j4
make PREFIX=. CC=$MPICC CXX=$MPICXX FC=$MPIF90 install -j4
export BLAS_LIB=$GPTUNEROOT/OpenBLAS/libopenblas.so
export LAPACK_LIB=$GPTUNEROOT/OpenBLAS/libopenblas.so
export LD_LIBRARY_PATH=$GPTUNEROOT/OpenBLAS/:$LD_LIBRARY_PATH
```

2.2.4 OpenMPI

OpenMPI is an open-source Message Passing Interface (MPI) implementation for distributed computations. To use the full feature of GPTune, one needs OpenMPI (4.0.1 or higher). Other MPI vendors such as CrayMPICH or SpectrumMPI can also be used, but only a limited number of GPTune features can be supported. We recommend the user install OpenMPI from source if the user's system does not have it installed yet.

```
cd $GPTUNEROOT
wget https://download.open-mpi.org/release/open-mpi/v4.0/openmpi-4.0.1.tar.bz2
bzip2 -d openmpi-4.0.1.tar.bz2
tar -xvf openmpi-4.0.1.tar
cd openmpi-4.0.1/
./configure --prefix=$PWD --enable-mpi-interface-warning --enable-shared --enable-
    static --enable-cxx-exceptions CC=$MPICC CXX=$MPICXX F77=$MPIF90 FC=$MPIF90 --
    enable-mpi1-compatibility --disable-dlopen
make -j4
make install
export PATH=$PATH:$GPTUNEROOT/openmpi-4.0.1/bin
export MPICC="$GPTUNEROOT/openmpi-4.0.1/bin/mpicc"
export MPICXX="$GPTUNEROOT/openmpi-4.0.1/bin/mpicxx"
export MPIF90="$GPTUNEROOT/openmpi-4.0.1/bin/mpif90"
export LD_LIBRARY_PATH=$GPTUNEROOT/openmpi-4.0.1/lib:$LD_LIBRARY_PATH
export LIBRARY_PATH=$GPTUNEROOT/openmpi-4.0.1/lib:$LIBRARY_PATH
```

It is highly recommended that one install OpenMPI from source as above, but if one wants to use an already installed MPI library (OpenMPI, CrayMPICH, SpectrumMPI), set the following accordingly:

```
export PATH=$PATH:XXX
export MPICC=XXX
export MPICXX=XXX
export MPIF90=XXX
export LIBRARY_PATH=XXX:$LIBRARY_PATH
```

2.2.5 ScaLAPACK

One needs to ensure that the ScaLAPACK library uses the same MPI version as in 2.2.4. If one installed OpenMPI from source, then very likely one has to also install ScaLAPACK from source:

```
cd $GPTUNEROOT
wget http://www.netlib.org/scalapack/scalapack-2.1.0.tgz
tar -xf scalapack-2.1.0.tgz
cd scalapack-2.1.0
rm -rf build
mkdir -p build
cd build
mkdir -p install
cmake .. \
-DBUILD_SHARED_LIBS=ON \
-DCMAKE_C_COMPILER=$MPICC \
-DCMAKE_Fortran_COMPILER=$MPIF90 \
-DCMAKE_INSTALL_PREFIX=./install \
-DCMAKE_BUILD_TYPE=Release \
-DCMAKE_VERBOSE_MAKEFILE:BOOL=ON \
-DCMAKE_Fortran_FLAGS="-fopenmp" \
-DBLAS_LIBRARIES="$BLAS_LIB" \
-DLAPACK_LIBRARIES="$LAPACK_LIB"
make
make install
export SCALAPACK_LIB="$PWD/install/lib/libscalapack.so"
```

2.2.6 Python packages in the requirement file

The following Python packages listed in requirement.txt can be installed automatically with pip: `numpy`, `joblib`, `scikit-learn`, `scipy`, `statsmodels`, `pyaml`, `matplotlib`, `GPy`, `openturns`, `lhsmdu`, `ipyparallel`, `opentuner`, `hpbandster`, `pygmo`, `filelock`, `requests`, `pymoo`, `cloudpickle`, `scikit-optimize`, `cGP`, `GPTuneHybrid`, and `autotune`..

```
env CC=$MPICC pip install --user -r requirements.txt
```

2.2.7 GPTune C code

```
cd $GPTUNEROOT
mkdir -p build
cd build
cmake .. \
-DBUILD_SHARED_LIBS=ON \
-DCMAKE_CXX_COMPILER=$MPICXX \
-DCMAKE_C_COMPILER=$MPICC \
-DCMAKE_Fortran_COMPILER=$MPIF90 \
-DTPL_BLAS_LIBRARIES=$BLAS_LIB\
-DTPL_LAPACK_LIBRARIES=$LAPACK_LIB \
-DTPL_SCALAPACK_LIBRARIES=$SCALAPACK_LIB \
-DGPTUNE_INSTALL_PATH=$SITE_PACKAGES_PATH
make
```

2.2.8 mpi4py

The mpi4py [7] is a Python interface for the MPI programming purpose, which provides an object-oriented approach to MPI bindings. The package allows us to translate the MPI syntax and semantics of standard MPI bindings for C++ to Python semantics.

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

2.2.9 GPTune Examples (SuperLU_DIST)

One can optionally install an application example, SuperLU_DIST, via the following commands. See the example scripts (search for "BuildExample") in 2.3 on how to build other application examples.

```
cd $GPTUNEROOT/examples/SuperLU_DIST
git clone https://github.com/xiaoyeli/superlu_dist.git
cd superlu_dist
mkdir -p build
cd build
cmake .. \
-DCMAKE_CXX_FLAGS="-Ofast -std=c++11" \
-DCMAKE_C_FLAGS="-std=c11 -DPRNTlevel=0 -DPROFlevel=0 -DDEBUGlevel=0" \
-DBUILD_SHARED_LIBS=OFF \
-DCMAKE_CXX_COMPILER=$MPICXX \
-DCMAKE_C_COMPILER=$MPICC \
-DCMAKE_Fortran_COMPILER=$MPIF90 \
-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
```

2.2.10 Test the installation (full version)

The basic functionality of GPTune can be tested using a synthetic function in examples/GPTune-Demo.

Set the tuning meta description. First, edit $GPTUNEROOT/run_env.sh to define machine name, processor architecture, number of compute nodes, number cores per node, and software dependencies. Take MacOS for example:

```
machine=mac #machine name
proc=intel      # processor type
mpi=openmpi  # mpi vendor
compiler=gnu  # compiler
nodes=1  # number of nodes
cores=8  # cores per node
```

```
# The following adds a few environment variables at runtime
export PATH=$PWD/openmpi-4.0.1/bin:$PATH
export MPIRUN="$PWD/openmpi-4.0.1/bin/mpirun"
export LD_LIBRARY_PATH=$PWD/openmpi-4.0.1/lib:$LD_LIBRARY_PATH
export DYLD_LIBRARY_PATH=$PWD/openmpi-4.0.1/lib:$DYLD_LIBRARY_PATH

# The following defines the metadata regarding the software dependency and machine information
software_json=$(echo ",\"software_configuration\":{\"openmpi\":{\"version_split\": [4,0,1]},\"scalapack\":{\"version_split\": [2,1,0]},\"gcc\":{\"version_split\": [10,2,0]}}")
loadable_software_json=$(echo ",\"loadable_software_configurations\":{\"openmpi\":{\"version_split\": [4,0,1]},\"scalapack\":{\"version_split\": [2,1,0]},\"gcc\":{\"version_split\": [10,2,0]}}")
machine_json=$(echo ",\"machine_configuration\":{\"machine_name\":\"$machine\",\"$proc\":{\"nodes\":$nodes,\"cores\":$cores}}")
loadable_machine_json=$(echo ",\"loadable_machine_configurations\":{\"$machine\":{\"$proc\":{\"nodes\":$nodes,\"cores\":$cores}}}")
```

Invoke the tuning experiments Once the run_env.sh has been set up properly, one can use the following to perform the selected tuning experiment.

```
cd $GPTUNEROOT
. run_env.sh
cd examples/GPTune-Demo
tp=GPTune-Demo # application name
app_json=$(echo "{\"tuning_problem_name\":\"$tp\"")
mkdir -p .gptune
echo "$app_json$machine_json$software_json$loadable_machine_json$loadable_software_json}" | jq '.' > .gptune/meta.json # generate the JSON file defining the meta data
$MPIRUN -n 1 python ./demo.py #invoke the tuning experiment
```

2.3 Installation using example scripts

The following example build scripts are available for a number of popular (HPC) systems. First, obtain GPTune from its GitHub repository:

```
git clone https://github.com/gptune/GPTune.git
cd GPTune/example_scripts
```

Second, users can directly use the following scripts depending on their systems.

- **Ubuntu/Debian-like systems supporting apt-get**. The following script installs everything from scratch and can take up to 2 hours depending on the users' machine specifications. If "MPIFromSource=0", you need to set PATH, LIBRARY_PATH, LD_LIBRARY_PATH and MPI compiler wrappers when prompted.

  ```
  bash config_cleanlinux.sh
  ```

- **macOS supporting homebrew.** The following script installs everything from scratch and can take up to 2 hours depending on the users' machine specifications. The user may need to

set pythonversion, gccversion, openblasversion, lapackversion on the top of the script to the versions supported by your homebrew software.

```
zsh config_macbook.zsh
```

- **NERSC Cori**. The following script installs GPTune with MPI, Python, compiler and cmake modules on Cori. Note that you can set "proc=haswell or knl", "mpi=openmpi or craympich" and "compiler=gnu or intel". Setting mpi=craympich will limit certain features of GPTune. In particular, only the so-called Reverse Communication Interface (RCI) mode and the lite mode can be used, see Section 3.2.2 for more details.

```
bash config_cori.sh
```

- **NERSC Perlmutter**. The following script installs GPTune with MPI, Python, compiler, cudatoolkit and cmake modules on Perlmutter. Note that you need to set "proc=milan #(CPU nodes) or gpu #(GPU nodes)", "mpi=openmpi or craympich" and "compiler=gnu". Setting mpi=craympich will only support the RCI mode and the lite mode.

```
bash config_perlmutter.sh
```

- **OLCF Summit**. The following script installs GPTune with MPI, Python, compiler, cuda and cmake modules on Summit. Note that you can set "proc=power9", "mpi=spectrummpi" and "compiler=gnu". Currently, only the RCI mode and the lite mode can be used on Summit.

```
bash config_summit.sh
```

- **OLCF Crusher**. The following script installs GPTune with MPI, Python, compiler, cuda and cmake modules on Crusher. Currently, only the RCI mode and the lite mode can be used on Crusher.

```
bash config_crusher.sh
```

In all the examples scripts above, setting "BuildExample=0" will only install GPTune with a ScaLAPACK *PDGEQRF* tuning example. Otherwise, all tuning examples (SuperLU_DIST, Hypre, STRUMPACK, ButterflyPACK, MFEM) will be installed.

The basic functionality of GPTune can be tested using a synthetic function under examples/GPTune-Demo.

Set the tuning meta description. First, edit $GPTUNEROOT/run_env.sh to define machine name, processor architecture, number of compute nodes, number cores per node, and software dependencies. Note that when GPTune is installed with the example scripts of Section 2.3, the run_env.sh already contains all the required runtime information for those supported machines. One simply needs to define **machine**, **proc**, **mpi**, **compiler** on top of run_env.sh to match those used in the config_*.sh file, and set **nodes** to desired number of compute nodes on those machines. For example, for any macOS system that has used config_macbook.zsh to build GPTune, one can use the following in run_env.sh.

```
export machine=mac
export proc=intel
export mpi=openmpi
export compiler=gnu
export nodes=1
```

Invoke the tuning experiments Once the run_env.sh has been set properly, one can use the following to invoke the tuning experiment.

```
cd $GPTUNEROOT/examples/GPTune-Demo
bash run_examples.sh # assuming run_env.sh has been modified properly
```

Note that there is a run_examples.sh file in each example folder examples/[APP]/run_examples.sh. One can test other applications by using that file accordingly.

2.4 Installation using Nix

Nix [9] is package manager and build system which can be used to both install GPTune's dependencies and build GPTune itself. Nix runs on most Unix-like systems - MacOS and Linux (including Windows Subsystem for Linux (WSL)) systems on both x86 and ARM. Installing GPTune with Nix will automatically install all of GPTune's dependencies as well; these dependencies are self-contained and will neither affect nor be affected by any other package on your system. Furthermore, as most of GPTune's dependencies are standard Nix packages which have been precompiled on the public NixOS binary caches, setup time should be well under an hour.

As a result, we recommend using Nix for installing GPTune on single-node systems - both personal computers and general server or cloud systems (even if you do not have root access). However, as Nix installs its own dependencies in a self-contained manner, Nix-based installs will not interact with system packages; as a result, if certain pre-installed packages are patched or given special privileges on a particular system, Nix will be unable to use them by default. As a result, the Nix-based installation is often not compatible out of the box with HPC systems where this is the case, such as US Department of Energy supercomputers.

Warning: If using Nix, please *do not manually modify or delete the contents of the nix store* (`/nix/store`, or `~/.nix/store` for unprivileged installs). Nix expects the contents of packages in the Nix store to be immutable, and modifications to the store will likely cause mysterious errors. If you wish to delete older versions of GPTune or its dependencies to save disk space, use the `nix-collect-garbage` command[1]. If such manual modifications have already occurred and Nix begins to complain about mismatched hashes or missing paths, pass the offending paths to `nix store repair`[2].

2.4.1 Install Nix

We provide these instructions for installing Nix for user convenience. Please note that the most detailed and up-to-date documentation for installing Nix can be found in the official NixOS documentation[3], and questions on installing Nix should be directed to the NixOS forums or chatrooms[4].

[1] https://nixos.org/manual/nix/stable/package-management/garbage-collection.html

[2] https://nixos.org/manual/nix/stable/command-ref/new-cli/nix3-store-repair.html

[3] https://nixos.org/download.html, https://nixos.org/manual/nix/stable/installation/installation.html

[4] https://nixos.wiki/wiki/Get_In_Touch

If you have root access, first ensure that the `LD_LIBRARY_FLAGS` and `LD_PRELOAD` flags are environment variables are not present in your environment:

```
unset LD_LIBRARY_PATH LD_PRELOAD
```

Then, run this command to automatically install Nix, then immediately proceed to Section 2.4.2:

```
sh <(curl -L https://nixos.org/nix/install) --daemon
```

Windows Subsystem for Linux (WSL) users should use the `--no-daemon` flag instead of `--daemon`.

Test the installation by running:

```
nix --version
```

If nix is not found, you need to restart the shell session.

If you do *not* have root access, you can install Nix as an unprivileged user. First, test if your system supports user namespaces, by running:

```
unshare --user --pid echo YES
```

and seeing if the string `YES` is printed. If the `unshare` command is missing, run the following commands (if one of the files is missing, try the other)

```
zgrep CONFIG_USER_NS /proc/config.gz
grep CONFIG_USER_NS /boot/config-$(uname -r)
```

If one of these commands prints `CONFIG_USER_NS=y`, then your system supports user namespaces. Many recent Debian, Ubuntu, and Arch Linux systems will have this support by default, but CentOS/RHEL systems and older systems will not. More information can be found at Nix's documentation on unprivileged installs[5].

If your system supports user namespaces, `nix-user-chroot` is the recommended installation method. To install it, first download the appropriate static binary for your hardware platform from the `nix-user-chroot` releases page[6]:

```
#replace the link below with the appropriate build for your architecture
wget -O nix-user-chroot https://github.com/nix-community/nix-user-chroot/releases/
    download/1.2.2/nix-user-chroot-bin-1.2.2-x86_64-unknown-linux-musl
chmod +x nix-user-chroot
#optionally, add nix-user-chroot to PATH - you'll be running it a lot
```

Then, select an installation location. For this example, we will use `~/.nix`. Note that Nix will perform a significant amount of disk I/O to this location, so make sure that this directory is not located on a network drive (NFS, etc.) or builds may be slowed by up to an order of magnitude (for example, we recommend that UC Berkeldey Millennium cluster users use a folder in `/scratch`) or `/nscratch`) instead of one in `/home`). You may then install nix with:

```
mkdir -m 0755 ~/.nix
./nix-user-chroot ~/.nix bash -c "curl -L https://nixos.org/nix/install | bash"
```

You may now enter the nix chroot environment with the following command[7]:

```
./nix-user-chroot ~/.nix bash -l
```

[5] `https://nixos.wiki/wiki/Nix_Installation_Guide#Installing_without_root_permissions`

[6] `https://github.com/nix-community/nix-user-chroot/releases`

[7] Troubleshooting note: if you run into "out of space" errors during builds, set the `TMPDIR` environment variable when you run this command to a location on a disk with plenty of space, e.g. `TMPDIR=/scratch/tmp nix-user-chroot ~/.nix bash`

This works much like a python virtualenv or conda shell - it will drop you into a environment where the Nix package manager and tools you have installed with Nix are available. As with a python virtualenv, you must be inside this environment in order to access tools (e.g. GPTune) that are installed with Nix (i.e. you must run it in each shell where you need these tools). All programs, files, etc. outside the environment should be accessible from within the environment as well.

On the other hand, **if you do *not* have root access and your system does *not* support user namespaces**, you can install Nix using proot; see the corresponding section[8] in the Nix installation guide for details.

2.4.2 Enable nix flakes

Nix flakes, which we use to build GPTune, are technically an experimental feature in Nix (this is not a mark of instability - flakes have existed and been widely used for years, but remain marked as experimental since there's a slim possibility that their interface might change). As a result, they must be manually enabled, which can be done by adding this line:

```
experimental-features = nix-command flakes
```

to any one (or more) of the following locations (if the file(s) in question don't exist, feel free to create them):

- `~/.config/nix/nix.conf` (recommended, affects your user account only)
- `/etc/nix/nix.conf` (if you have root access, and want to enable flakes for everyone on your system)
- `/nix/etc/nix/nix.conf` (for `nix-user-chroot` installs only; this file should **only** be modified from within the chroot environment)

2.4.3 Build GPTune

Clone the GPTune repo and cd into its directory:

```
git clone https://github.com/gptune/GPTune
cd GPTune
```

then run

```
nix develop
```

to enter an environment where the `python` executable has all the dependencies needed, and set the `$GPTUNEROOT` environment variable to GPTune's installation path in the Nix store. The `nix develop` environment is similar to a Python virtual environment: it must be loaded every time GPTune or its dependencies are run.

2.4.4 Test the installation

The basic functionality of GPTune can be tested using a synthetic function, in examples/GPTune-Demo. First, modify the machine name, processor architecture, number of compute nodes, number of cores per node, and software dependencies in the meta json file at

[8] `https://nixos.wiki/wiki/Nix_Installation_Guide#PRoot`

```
$GPTUNEROOT/examples/GPTune-Demo/.gptune/meta.json
```

Then one can invoke the basic tuning example via

```
cd $GPTUNEROOT/examples/GPTune-Demo
mpirun -n 1 python ./demo.py -nrun 20 # 20 is the total number of function
    evaluations
```

2.5 Installation from Spack

One can also consider using Spack[9], a package manager that can automatically install GPTune and its dependencies. To install and test GPTune using Spack (the develop branch of the spack GitHub repo is highly recommended), one simply needs one of the following commands. Note that if this is the first time the user uses Spack, the following commands can take more than 10 hours to install everything from scratch.

```
spack install gptune@master~mpispawn  # only install the RCI interface
spack install gptune@master  # install both the RCI and spawning interfaces
spack install gptune@master+hypre  # install the hypre example application
    together with gptune
spack install gptune@master+superlu  # install the superlu-dist example
    application together with gptune
```

For GPTune installed on Spack, you can test its functionality and use it for your own applications by:

Test the Spack GPTune module Try the following lines to validate the Spack installation.

```
spack load gptune # load modules and set environment variables
spack test run gptune # test the spack gptune module
see ~/.spack/test/xxx/gptune-xxx-test-out.txt for the test run log
```

Use the Spack GPTune module for external applications Once you have run the spack test run gptune, a run_env.sh file will be generated that includes all the information required to use the Spack GPTune package by an external application, located at

```
[SPACK_ROOT]/opt/spack/[system info]/[compiler version]/gptune-xxx/.spack/test/
    run_env.sh
```

Simply copy it to any place you want and run

```
. run_env.sh
```

2.6 Docker image

If the user does not want to install GPTune in the user's local user environment, a Docker image is available for all system types: Linux, Mac or Windows. First install and launch docker following the instructions at https://docs.docker.com. The Docker image can then be obtained and launched by

```
docker pull liuyangzhuan/gptune:4.5
```

[9] https://spack.io/

To test the pre-built Docker image, simply try:

```
docker run -it -v $HOME:$HOME liuyangzhuan/gptune:4.5 # launch the image
edit run_env.sh # change lines 53 and 418 to the number of nodes and cores per
    node in your system
cd $GPTUNEROOT/examples/GPTune-Demo
bash run_examples.sh
```

In addition to the GPTune-Demo example, one can try all application examples included in the Docker image to demonstrate advanced GPTune features, using the following instructions:

- Part I: Basic usage of GPTune: https://bit.ly/3kbeA7p
- Part II: Advanced algorithms of GPTune: https://bit.ly/37JylQH
- Part III: History database and crowd-tuning: https://bit.ly/37Jn4zZ

3 GPTune Implementation

This section describes how GPTune is implemented with some details of the tuning algorithm. We refer users to our technical papers [37, 26, 41, 5, 28, 27] for a more detailed exposition of the methodology.

3.1 Algorithms

We first describe the algorithms for single- and multi-objective autotuning with single- and multi-task learning features. Then, we describe algorithms for GPTune's other autotuning features such as transfer learning, incorporating performance models like clustered GP, hybrid models, multi-fidelity tuning, and sensitivity analysis.

As outlined in Section 1, GPTune uses Bayesian optimization, where we iteratively build and update a GP-based surrogate model by evaluating the objective functions for the chosen parameter configurations. Using the notation described in Table 2, we optimize a given objective function, 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 configuration of the tuning parameters. $\mathbb{IS}$ is the *Task Parameter Input Space* that contains all the input problems that the application may encounter, and $\mathbb{PS}$ is the *Tuning Parameter Space* that defines the parameter search region. GPTune handles three different types of variables, *Real*, *Integer*, and *Categorical*.

- *Real variables* (a.k.a. continuous variables) have (in theory) uncountably many values and a natural intrinsic ordering of these values, similar to real numbers $\mathbb{R}$.
- *Integer variables* (a.k.a. discrete variables, or ordinal variables) have multiple value categories and with a natural intrinsic ordering on these categories, like integer numbers $\mathbb{Z}$.
- *Categorical variables* (a.k.a. nominal variables) have multiple value categories, but there is no natural intrinsic ordering on the categories.

GPTune's tuning algorithms (except GPTuneHybrid) internally convert these three types of variables into real variables with range $[0,1]$ for surrogate modeling and parameter search. GPTuneHybrid (Section 3.1.6) uses a tree-based search for categorical variables (conversion not needed) and continuous GP for real/integer variables (conversion needed).

3.1.1 Single-task and multi-task learning (single-objective)

This section describes GPTune's main tuning algorithm for a single-objective tuning problem. It comprises a multi-task learning autotuning (MLA) feature which can tune multiple tasks simultaneously (e.g., tuning ScaLAPACK's PDGEQRF for multiple different matrix dimensions) while sharing common knowledge between then in order to improve the accuracy of the surrogate model. If the user supplies only one tuning task, we refer to it as single-task learning autotuning (SLA) to distinguish it from MLA.

The tuning process consists of the following phases:

1. **Sampling phase.** There are two (optional) sampling steps.

   - The first is to select a set T of n_i tasks $T = [t_1; t_2; \ldots; t_{n_i}] \in \mathbb{IS}^{n_i}$. T is a list of target tasks specified by the user. If the user supplies T, then there is no need to perform any sampling for T. Otherwise, GPTune can perform sampling to get a representative set of input tasks that the application may encounter, via random sampling.
   - The second sampling step is to select an initial set $X = [X_1; X_2; \ldots; X_{n_i}]$ of tuning parameter configurations for every task. These pilot samples are used to randomly sample the tuning parameter space to guide the generation of samples proposed by the surrogate model. Using the notation in Table 2, let NS denote a prescribed total number of function evaluations per task. The number of initial samples is set to NS1, with NS1 = $\lfloor$NS/2$\rfloor$ by default. Users can also set a specific NS1 value, i.e., $NS1 \neq \lfloor$NS/2$\rfloor$. If $NS1$ is given as 0, we do not perform any pilot sampling and start directly with a surrogate-based search. The user can also load historical function evaluations when starting the tuner. In this case, assuming NS_{hist} refers to the number of function evaluations loaded for each of the given tasks, we perform an additional initial random sampling of $NS1$-NS_{hist} samples, if $NS1 > NS_{hist}$. Let n_s denote the number of samples available per task. For task t_i, its initial sampling X_i consists of n_s = NS1 tuning parameter configurations $X_i = [x_{i,j}]_{j\in[1,n_s]} \in \mathbb{PS}^{n_s}$. Define $X = [X_1; X_2; \ldots; X_{n_i}] \in \mathbb{PS}^{n_i \times n_s}$ to represent all samples.

   The samples $x_{i,j}$ are evaluated through the runs of the application, whose results, $y_{i,j} = y(t_i, x_{i,j}) \in \mathbb{OS}$, can be denoted by $Y_i = [y_{i,j}]_{j\in[1,n_s]} \in \mathbb{OS}^{n_s}$. The set $Y = [Y_1; Y_2; \ldots; Y_{n_i}] \in \mathbb{OS}^{n_i \times n_s}$ represents the results of all these evaluations.

   Relevant GPTune parameters. NI (or ntask): number of tasks (i.e., n_i). Note that we use NI as the code notation and n_i as the algorithm notation. Tgiven: (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_max_iter, sample_random_seed, which are described in Section 6.1.10.

2. **Modeling phase.** This phase builds a Bayesian posterior probability distribution of the objective function by 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 a *Gaussian Process* (GP) in the multi-output setting. If there is only one tuning task, we use a standard GP model.

- **Standard GP**. A GP represents a probability model $f(x)$ for the objective function $y(x) = f(x) + \varepsilon$, ε is a zero-mean Gaussian noise. It assumes that $(f(x_1), ..., f(x_{n_s}))$ is jointly Gaussian, with a 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 considered close 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: $f(x) \sim GP(\mu(x), \Sigma)$.

  By default, $\mu(x)$ is initialized as the zero function [33]. The modeling is done through $\Sigma(X, X)$, by maximizing the log-likelihood of the samples X with values Y in GP. For single task learning, the size of the covariance matrix $\Sigma(X, X)$ is $n_s \times n_s$.

- **Linear Coregionalization GP Model (LCM)**. 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 the 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 $a_{i,q}$ ($i \in [1, n_i]$) 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 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 that the covariance of the latent function is based on a Gaussian kernel. Note that GPTune also supports other kernels in addition to Gaussian:

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

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

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

  where $\delta_{i,j}$ is the Kronecker delta function, $b_{i,q}$ and d_i are diagonal regularization parameters.

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 the 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) [25] through the Python package scikit-Optimize [18]. 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 [17] to enable distributed memory parallel modeling, which is useful when the model size is very large.

Relevant GPTune parameters. model_latent: number of latent functions (i.e., Q). By default $Q = n_i$. model_kern: type of kernel K, 'RBF' by default. The other relevant options are model_class, model_output_constraint, model_threads, model_processes, model_restarts, model_restart_processes, model_restart_threads, model_max_iters, model_jitter, model_max_jitter_try, model_max_iters, model_sparse, model_random_seed, etc. See Section 6.1.10 for details.

Algorithm 1 Bayesian optimization-based single-objective MLA

1: **Sampling phase:** Compute $y(t_i, x)$, $i \leq n_i$ at NS1 initial random tuning parameter configurations for n_i selected tasks. Set $n_s =$ NS1.
2: **while** $n_s \leq$ NS **do**
3: **Modeling phase:** Update the hyperparameters in the LCM model of $y(t_i, x)$, $i \leq n_i$ using all available data.
4: **Search phase:** Search for an optimizer x_i^* for the EI of task t_i, $i \leq n_i$. Let $X^* = [x_1^*; x_2^*; \ldots; x_{n_i}^*]$.
5: Compute $y(t_i, x)$, $i \leq n_i$ at the new tuning parameter configurations X^*.
6: $n_s \leftarrow n_s + 1$.
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 values of the objective function at the new points $X^* = [x_1^*; x_2^*; \ldots; x_{n_i}^*]$ can be predicted with the posterior mean $\mu_* = [\mu_1^*; \mu_2^*; \ldots; \mu_{n_i}^*]$ and the posterior variance (prediction confidence) $\sigma_*^2 = [\sigma_1^{*2}; \sigma_2^{*2}; \ldots; \sigma_{n_i}^{*2}]$ 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 [12]. Mean and variance can be used to construct the Expected Improvement (EI) acquisition function, which can be maximized to choose a new point X^* for function evaluation (additional sampling). GPTune supports multiple Python packages to optimize the acquisition function, including PyGMO [2], pymoo [4], and SciPy [40]. By default, we use the particle swarm optimization algorithm (PSO) [22] in PyGMO if the full version of GPTune is installed, or the Trust-Region Constrained Algorithm in SciPy if the lite version of GPTune is installed.

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

Algorithm 2 Bayesian optimization-based multi-objective MLA

1: **Sampling phase:** Compute $y^l(t_i, x)$, $i \leq n_i$, $l \leq \gamma$ at NS1 initial random tuning parameter configurations for n_i selected tasks. Set $n_s =$ NS1.
2: **while** $n_s \leq$ NS **do**
3: **Modeling phase:** For each objective $l \leq \gamma$, update the hyperparameters in the LCM model of $y^l(t_i, x)$, $i \leq n_i$ using all available data.
4: **Search phase:** Search for k best tuning parameter configurations for the EI of task t_i, $i \leq n_i$.
5: Compute $y^l(t_i, x)$, $i \leq n_i$ at k new tuning parameter configurations.
6: $n_s \leftarrow n_s + k$.
7: **end while**
8: Return the Pareto set parameter configurations and their objective function values for each task.

Relevant GPTune parameters. search_class: Python packages for the search. search_algo: evolutionary algorithms supported by each package. By default search_algo is 'pso' from PyGMO. The other relevant options are search_pop_size, search_gen, search_evolve, search_max_iters, search_more_samples, search_random_seed, search_algo, etc. See Section 6.1.10 for details.

3.1.2 Single-task and multi-task learning (multi-objective)

The MLA algorithm described in Section 3.1.1 can be easily extended to multi-objective, multi-task settings, as shown in Algorithm 2. Algorithm 2 describes the multi-objective extension of Algorithm 1. Let $y^l(t, x)$, $l \leq \gamma$ denote the lth objective function. The algorithm essentially builds an LCM model per objective function $y^l(t, x)$ in the modeling phase. In addition, the search phase relies on multi-objective evolutionary algorithms such as the non-dominated sorting generic algorithm II (NSGA-II) [8] to search for k new tuning parameter configurations in each iteration. Sorting among all possible sequential candidates is based on Pareto dominance and crowding distance [8].

More specifically, this unique approach of GPTune consists of two phases and is well suited for budget-limited tuning and optimization tasks. The first phase is updating the surrogate model taking multi-task setting into consideration. The second phase is to come up with candidates for sequential sampling, where the main difference between single-objective and multi-objective optimization lies. In each step of Algorithm 2, surrogates are modeled with the MLA model. Then, the Pareto sets of these surrogates are obtained by a chosen heuristic algorithm (e.g., NSGA-II Based on the resulting Pareto set, we select a batch (for size k which is a GPTune parameter "search_more_samples") of sequential sample candidates for subsequent model updates. See more details in [26].

GPTune also allows users to specify a certain output constraint for each of the objectives. For example, for a tuning problem concerning both runtime and accuracy, one might want to minimize the runtime while satisfying a certain level of accuracy without needing to maximize accuracy (i.e., minimize the error). In GPTune, for each objective, users can specify whether they want to optimize the objective, optimize the objective only within a desired range, or just want to make sure that the objective is within the given desired range (the detailed user interface is given in Section 6.1.6 and an example is given in Section 4.2.3). More specifically, assuming there are two objectives (y^1 and y^2), GPTune can handle the following possible scenarios:

- Minimize y^1 and y^2 with no output constraints: GPTune runs Algorithm 2 and tries to find a Pareto front.
- Minimize y^1 and y^2 with a range constraint on y^1: After a function evaluation, if the output of y^1 for the parameter configuration is outside the given range constraint, then the parameter configuration is not used when training the surrogate model. GPTune inserts a large value for the parameter point when fitting the surrogate model, so that GPTune would not tend to explore parameter configurations near the out-of-the-range parameter configuration.
- Minimize y^2 only while satisfying an output constraint on y^1: We only build a surrogate model for y^2 which we are aiming to optimize. We do not build a surrogate model for the objective y^1. If y^1's output is within the desired range, we use the parameter configuration for surrogate modeling, otherwise we use a large number for the parameter configuration instead of the actual output value. If there is only one objective to optimize, we can use PSO instead of NSGA-II.

Relevant GPTune parameters. search_class: the Python package for the multi-objective search: PyGMO (default in the full version of GPTune) or pymoo (default in the lite version of GPTune). search_algo: evolutionary algorithms. By default, search_algo is 'nsga2' (i.e., the NSGA-II algorithm) from PyGMO. search_more_samples: the number of additional samples per iteration (i.e., k). Default to 1. The other relevant options are search_pop_size, search_gen, search_evolve, search_max_iters, etc. See Section 6.1.10 for details.

3.1.3 Incorporation of cheap performance models

A cheap performance model refers to an analytical formula or an inexpensive application run (can be run at least thousands of times in reasonably short time) for any feature (e.g., 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 runtime. When available, performance models can be incorporated to build a more accurate LCM model with fewer samples needed. In what follows, the incorporation of the performance model is explained by assuming a single task $n_i = 1$ and a 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 the parameter configuration x.

In GPTune, we can enrich the observed data with the performance model $\tilde{y}$ and build a more informative surrogate model. Without the performance model, the entries in the LCM kernel matrix represent the covariance between the points x and x' in the feature space $\mathbb{PS}$ of dimension β. The values $\tilde{y}(x)$ can be used as extra features to construct an enriched feature space of dimension $\beta + \tilde{\gamma}$ consisting of points $[x, \tilde{y}(x)]$. Note that the enriched LCM matrix still has the same dimension $n_s n_i \times n_s n_i$. 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 $[x^*, \tilde{y}(x^*)]$. In this approach, the surrogate model would capture the observed model and the analytic model as two outputs. Instead of modeling the performance model directly, this would allow us to leverage the correlation between the observed and analytic models.

3.1.4 Transfer learning approaches (Type I and Type II)

Transfer Learning-based Autotuning (TLA) focuses on predicting the performance of a specified *target task* indexed by t^*, with the knowledge of the performance data and surrogate models previously collected for n_i other (but correlated) *source tasks* $t_j, j \leq n_i$. In the typical TLA setting, one is allowed to collect new performance data for the target task, but no more performance data can be generated for the source tasks. Intuitively, the TLA model is capable of inferring the unobserved target task to an extent, with extrapolation of prediction using the information learned from the source tasks. GPTune offers two different types of TLA approaches.

Type I: Build an improved surrogate for the target task function $(x, y(t^*, x))$**.** The Type I approach focuses on building a surrogate model for the target task with existing data obtained from different source tasks and subsequent data obtained from the target task. The intuition is that we can leverage the kernel correlation between existing (source) and target tasks to construct a more accurate surrogate for the target task. As a result, TLA (Type I) allows us to achieve the optimal tuning parameters for the target task with fewer function evaluations, compared to a non-TLA approach that does not leverage information from the source tasks.

GPTune offers several TLA methods which include a number of existing methods from the literature [29, 16] as well as GPTune's novel ideas. The supported methods are outlined as follows:

- Multi-task learning-based TLA [37] (`TLA_method=LCM` or `TLA_method=LCM_BF`): This approach treats TLA as MLA, and handles the surrogate modeling within the MLA framework, for tuning the new task with the available data from the source tasks. The method exploits observed function evaluation data and/or pre-trained surrogate performance models of source tasks and runs the LCM-based multi-task surrogate modeling [37] (in Section 3.1.1) for both source and target tasks. It starts with building an LCM model with available data for the source tasks and no sample for the target task. The LCM model will sequentially obtain samples only from the target task. The new sample is then evaluated, and the LCM model is updated with one more sample from the target task. Eventually, the joint LCM model is built for both the source and target tasks.

  To provide data of the source tasks, one can supply all the available function evaluations of the source tasks (we call it `TLA_method=LCM`) or one can supply a pre-trained surrogate model of each of the source tasks as a black-box function (`TLA_method=LCM_BF`).

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

$$\mu(x) = \mu_{target}(x) + \sum_{i=1}^{n_{src}} \mu_{src_i}(x) \tag{7}$$

$$\sigma(x) = (\sigma_{target}(x) \cdot \Pi_{i=1}^{n_{src}} \sigma_{src_i}(x))^{\frac{1}{n_{src}+1}} \tag{8}$$

  where $\mu(x)$ and $\sigma(x)$ represents the mean and standard deviation function predicted by the surrogate model for the given tuning parameter configuration x. μ_{src} and σ_{src} are the mean

and standard deviation functions of the surrogate model of the n_{src} source task's surrogate models. μ_{target} and σ_{target} are the mean and standard deviation functions of the target task.

The idea of using a sum of surrogate models has been employed in another autotuning framework HiPerBOt [29], where the authors mention that the surrogate models of the source and target tasks can have different weights, however they do not provide a generalized approach to computing the weight values.

- Weighted average of surrogate models (`TLA_method=Regression`): GPTune provides a generalized weighted-sum approach to compute appropriate weight values for the surrogate models. This method `Regression`, assigns weights for each of the surrogate models and performs a linear regression-based method to compute the weights.

$$\mu(x) = w_{target} \cdot \mu_{target}(x) + \sum_{i=1}^{n_{src}} w_{src_i} \cdot \mu_{src_i}(x) \tag{9}$$

$$\sigma(x) = (\sigma_{target}(x)^{w_{target}} \cdot \Pi_{i=1}^{n_{src}} (\sigma_{src_i}(x)^{w_{src_i}})) \tag{10}$$

where w_{src} and w_{target} are the weights for source and target tasks, respectively (sum of all the weights is normalized to 1).

To determine the weights, for an individual GP surrogate model (both source and target tasks), for each of the observed samples for the target task we compute the difference between the predicted output of the sample's parameter configuration and the predicted output for the best parameter configuration observed so far. Then, we compare the differences from the prediction to the differences from the actual observations (for the current target task). More specifically, considering the linear regression problem with coefficients $w_{src_1}, \ldots, w_{src_{n_s}}$ for the source surrogate models and coefficient w_{target} for the target task surrogate model, and the sequential samples observed for the target task $(x_1, y_1), \ldots, (x_{n_s}, y_{n_s})$, we use a linear regression for the following:

$$y^* - y_j = \sum_{i=1}^{n_{src}} w_{src_i} \cdot [\mu_{src_i}(x^*) - \mu_{src_i}(x_j)] + w_{target} \cdot [\mu_{target}(x^*) - \mu_{target}(x_j)], j = 1, \ldots, n_s \tag{11}$$

where $y^* = f(x^*)$ is the current minimum black-box function value, and μ_{src_i} and σ_{src_i} are the mean and standard deviation functions of the ith surrogate model among the n_{src} source task's surrogate models. μ_{target} and σ_{target} are the mean and standard deviation functions of the target task. We compute the weight values for individual surrogate models at each iteration of the tuning for both the target and the source tasks, and substitute the weights into the weighted sum equation (Equation 9) in the search phase. Note that, the LHS and RHS of the linear regression problem of Equation 11 are also normalized by y^* and $\mu_{src}(x^*)/\mu_{target}(x^*)$, because source and target tasks can have different output scales. The scaled expression is, therefore, given as follows:

$$\frac{y^*}{y^*} - \frac{y_j}{y^*} = \sum_{i=1}^{n_{src}} w_{src_i} \cdot [\frac{\mu_{src_i}(x^*)}{\mu_{src_i}(x^*)} - \frac{\mu_{src_i}(x_j)}{\mu_{src_i}(x^*)}] + w_{target} \cdot [\frac{\mu_{target}(x^*)}{\mu_{target}(x^*)} - \frac{\mu_{target}(x_j)}{\mu_{target}(x^*)}], j = 1, \ldots, n_s \tag{12}$$

- Stacking Gaussian Processes (`TLA_method=Stacking`): Unlike the additive combination approaches above, another transfer learning approach uses iterative model fitting to combine the source task with the surrogate model of the target task. In GPTune, we implemented this transfer learning algorithm presented in Google's Vizier [16] and allow users to use it from our unified tuning interface. The method starts by building a surrogate model of an initial source task. Then, for the second source task, it trains another GP model for the residual between the second source task and the initial surrogate model and combines the initial GP model with the residual model. It then repeatedly adds residual models for the remaining source tasks and the new task. The combined posterior mean is then computed as follows

$$\mu(x) = \mu'_{target}(x) + \sum_{i=1}^{n_{src}} \mu'_{src_i}(x) \tag{13}$$

μ'_i (when $2 \leq i \leq n_{src}$) is the mean function of the trained GP model for the residual between the observed samples for the task i and the predicted mean of μ'_{i-1}, as detailed in the following Equation 14.

$$\mu'_i(x) = \begin{cases} \text{GP mean trained with } \{(x_{src_{1,1}}, y_{src_{1,1}}), \ldots, (x_{src_{1,N}}, y_N^{src_1})\}, \text{ if } i = 1 \\ \text{GP mean trained with } \{(x_{src_{i,1}}, y_{src_{i,1}} - \sum_{j=1}^{i-1} \mu'_j(x_{src_{i,1}})), \ldots, \\ \quad (x_{src_{i,N}}, y_{src_{i,N}} - \sum_{j=1}^{i-1} \mu'_j(x_{src_{i,N}}))\}, \text{ if } 2 \leq i \leq n_{src} \\ \text{GP mean trained with } \{(x_{target_1}, y_{target_1} - \sum_{j=1}^{n_{src}} \mu'_j(x_{target_1})), \ldots, \\ \quad (x_{target_N}, y_{target_N} - \sum_{j=1}^{i-1} \mu'_j(x_{target_N}))\}, \text{ if } i = target \end{cases} \tag{14}$$

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

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

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

According to the Vizier paper [16], one would expect that the sequence (ordering) of the surrogate models of the source tasks can affect the quality of the combined model, and the first surrogate model should be well-trained. By default, we order the source surrogate models based on the number of collected samples (i.e., the first source task surrogate model contains the largest numbe of samples.

- Ensemble of TLA approaches: The best TLA method may change depending on the given tuning problem. While users can choose a TLA method from a unified interface, GPTune also provides several ensemble techniques in order to combine the benefits of different TLA methods.

  - Ensemble with toggling (`TLA_method=Ensemble_Toggling`): This method toggles a number of TLA approaches at each function evaluation, as detailed in Algorithm 3.
  - Ensemble with a probability function (`TLA_method=Ensemble_Prob`): This approach dynamically chooses one of the TLA approaches based on a probability distribution function drawn from the historical best performance, as detailed in Algorithm 4. For the

probability function, we assign a probability to each model, proportional to their best sample's inverse output value, since we are minimizing the objective.

Algorithm 3 Toggling TLA method

T = list of TLA approaches (e.g., T = { Regression, LCM, Stacking })
NS = tuning budget (#samples)
while $n_s \leq$ NS **do**
    TLA_method = $T[n_s \; mod |T|]$
    Update a TLA model using the chosen TLA method.
    Search x the next parameter configuration to evaluate.
    Compute $y(x)$ at the new parameter configuration x.
    $n_s \leftarrow n_s + 1$.
end while
Return X_{opt} and Y_{opt}

Algorithm 4 Selecting TLA method via a probability distribution function built from the best historical sample.

T = list of TLA approaches (e.g., T = { Regression, LCM, Stacking })
ExplorationRate = a user given rate for randomly choosing a TLA option
NS = tuning budget (#samples)
while $n_s \leq$ NS **do**
    if $rand() < ExplorationRate$ **then**
        TLA_method = Randomly choose one model from T
    else
        TLA_method = Probabilistically choose one model from T using the historical best sample from each TLA method. We assign a probability to each model, proportional to their best sample's inverse output value, since we are minimizing the objective.
    end if
    Update a TLA model using the chosen TLA method.
    Search x the next parameter configuration to evaluate.
    Compute $y(x)$ at the new parameter configuration x.
    $n_s \leftarrow n_s + 1$.
end while
Return X_{opt} and Y_{opt}

Type II: Build a surrogate for the target task function $(t, x_{opt}(t))$. The second class of approaches (named TLA_II), on the other hand, aims to predict the best parameter configuration of the new task without creating a surrogate model for the function $(x, y(t^*, x))$. Instead, we build surrogate models for the function $(t, x_{opt}(t))$ with $x_{opt}(t) = \arg\min_x f(t, x)$, i.e., the function from any (source or task) t to its optimal tuning parameter configuration. More specifically, letting $x_{opt}(t) = [x^1_{opt}(t), \ldots, x^\beta_{opt}(t)]$, we use a simple Gaussian Process to build a surrogate model for each $x^d_{opt}(t)$, $d = 1, \ldots, \beta$, using only the known values of $x^d_{opt}(t_j)$, $j \leq n_i$ from the source tasks. In other words, no surrogate model or objective function value is needed from the source tasks.

We expect to achieve a reasonably good parameter configuration for the new task with no need to collect any further data to build a surrogate. Once the optimal tuning parameters are predicted, one function evaluation on the new task using those parameters is performed. Note that unlike TLA (Type I), once the surrogate models are built, the models can be easily re-used without retraining for any unseen tasks t^*.

We provide a detailed user interface for the TLA_I and TLA_II approach in Section 6.2.5 and Section 6.2.6, respectively. We also explain database interfaces in Section 6.3 that help query relevant performance data for source tasks to be fed into the TLA methods.

Relevant GPTune parameters. TLA_method supported algorithms in the first type of TLA approach: `TLA_LCM`, `TLA_Regression`, etc. See Section 6.1.10 for details.

3.1.5 cGP for nonsmooth objectives

GPTune can provide improved tuning performance for applications that exhibit discontinuity or jumps in their objective functions, which are very common in many HPC application codes. The proposed GPTune variant is called clustered GP (cGP) [28], whose essential methodology is to leverage ML algorithms to detect the locations of the discontinuities and build a GP for each continuous partition. This allows us to separate potential different regimes of the black-box function based on the sequential samples.

More precisely, the proposed surrogate model uses the *decision boundaries* of classifiers trained by cluster results on observations to detect the partition boundaries and model different components (induced by non-smooth jumps or drops in the black-box function) with different additive GP components directly. The data used for building each model are determined by a supervised classification algorithm. The exploration of the next sequential sample is performed with a modified acquisition function weighted by the size of the clusters (and also the label counts in classification).

We introduce clustering based on the joint input-response pair $(\boldsymbol{x}, y(\boldsymbol{x}))$ in accordance with our sample-based definition of non-smoothness, and different combinations of clustering and classification algorithms would result in different types of additive schemes in the cGP kernels.

The innovation of cGP lies in the fact that we dynamically determine the non-smoothness regime based on the limited samples, and use both input and response for creating a non-stationary additive regime. Currently, the cGP can be used for arbitrary-dimensional input and univariate response output, but its multi-task generalization remains a future work.

Relevant GPTune parameters. N_PILOT_CGP: number of initial samples NS1. N_SEQUENTIAL_CGP: number of sequential samples NS-NS1. EXPLORATION_RATE_CGP: exploration rate is the probability (between 0 and 1) for accepting the proposed samples by the acquisition function. CLUSTER_METHOD_CGP: choice of the clustering algorithm. Other relevant parameters are METHOD_CGP, N_NEIGHBORS_CGP, N_COMPONENTS_CGP, ACQUISITION_CGP. See Section 6.1.10 for details.

3.1.6 GPTuneHybrid models for categorical and mixed variables

For tuning problems involving categorical or mixed-type tuning parameters, GPTune typically converts all parameter types to real values in $[0, 1]$ and builds GP models (see Section 3.1). However, several research papers have pointed out the unexpected consequences if categorical variables are not properly handled in the GP context [14]. We summarize the differences between three main kinds of tuning parameters in the table 3. Based on the different characteristics of each kind of

	Continuous var.	Integer var.	Categorical var.
Representation	$\mathbb{R}$	$\mathbb{Z}$	finite set
Magnitude	Yes	Yes	No
Order	Yes	Yes	No
Encoding	No	Yes	Yes
Gradient	continuous gradient	discrete gradient	No
Surrogate	GP	GP/Tree	Tree

Table 3: Comparison between different types of variables.

parameter, we support a variant of GPTune, called GPTuneHybrid [27], which relies on Monte Carlo Tree Search (MCTS) to guide the exploration of categorical part of the variables, and the GP to guide the exploration of continuous part. At each iteration we choose a more appropriate kernel that incorporates both categorical and continuous parts to model both parts jointly in a GP surrogate.

GPTuneHybrid takes three essential steps to generate the sequential samples: 1) heuristic MCTS search (with heuristic policies defined by policy_hybrid) to add new categorical samples, 2) dynamic kernel selection based on certain selection criteria (option selection_criterion_hybrid) to choose the best mixed-variable GP surrogate, 3) acquisition function (chosen by acquisition_GP_hybrid) maximization on the categorical value conditioned continuous GP surrogate to add new continuous samples. Currently, GPTuneHybrid can handle all types of input and output constraints as the other GPTune variants. However, it may require a large number of function evaluations for complex constraints.

Relevant GPTune parameters. selection_criterion_hybrid: model selection criteria. policy_hybrid: MCTS search policy. acquisition_GP_hybrid: GP search policy. n_budget_hybrid: number of total samples, n_find_leaf_hybrid: number of samples in GP (i.e., leaf node of the tree), n_pilot_hybrid: number of initial random samples. See Section 6.1.10 for details.

3.1.7 GPTuneBand for multi-fidelity autotuning

GPTune can leverage the LCM model to support multi-fidelity and multi-task tuning features. The variant is called GPTuneBand. Consider n_i tasks $y(t_i, x)$, $i \leq n_i$. For many applications, we can also evaluate modified tasks $y(t_i, x, b)$, $b \in [b_{\min}, b_{\max}]$, where $y(t_i, x, b_{\max}) = y(t_i, x)$ represents the true (highest-fidelity) objective and $y(t_i, x, b_{\min})$ represents the cheapest (lowest-fidelity) objective. Let's assume that the objective function is defined as "def objectives(point):" with a dictionary argument "point"; the multi-fidelity tuning algorithm allows the argument point to contain a key "budget" denoting the fidelity level, which can be mapped by the user to a code option, e.g., mesh sizes, tolerance in an Krylov solver, number of time steps in a PDE solver, etc.

GPTuneBand combines the LCM model with multi-armed bandit schemes by allowing $s_{\max}+1$ number of "arms", where $s_{\max} = \lfloor \log_\eta(b_{\max}/b_{\min}) \rfloor$. Each arm s starts with $N(s) = \left\lfloor \frac{s_{\max}+1}{s+1} \right\rfloor \eta^s$ number of samples at fidelity $B(s)$, which by default is $B(s) = b_{\max}\eta^{-s}$. Note that $B(s)$ corresponds to point["budget"] mentioned above. GPTuneBand leverages LCM to build a joint model across all arms and tasks to generate these samples. After the sampling is finished, each arm follows with a successive halving (SH) algorithm to recursively select the best $1/\eta$ samples in its arm and evaluate

them with an increased fidelity level, until the highest fidelity has been reached. This procedure is briefly summarized in Algorithm 5. Note that in practice, one can perform the GPTuneBand algorithm with multiple passes to generate more samples. For more details, please refer to [41].

Note that the GPTune also supports an interface to a multi-fidelity, single-task tuner HpBandSter (see Section 6.2.9), which does not support multi-task learning features. A detailed comparison of GPTuneBand with existing single-fidelity or single-task tuners can be found in [41].

Algorithm 5 GPTuneBand Algorithm (one pass) [41]

Inputs: fidelity function $B(\cdot)$, $s_{\max}$, η, tasks: $t_1, \ldots, t_{n_i}$

1: Compute the starting number of samples $N(s)$ for each bracket s:
2: $N(s) = \left\lfloor \frac{s_{\max}+1}{s+1} \right\rfloor \eta^s$ for $s = 0, 1, \ldots, s_{\max}$.
3: **for** $s \in \{0, \ldots, s_{\max}\}$ **do**
4: build the current LCM-task set $T = \{T_{ij} : i = 1, \ldots, n_i, j = s, \ldots, s_{\max}\}$, where an LCM-task T_{ij} means task t_i with starting fidelity $B(j)$.
5: build or update LCM$(|T|, N(s))^{\dagger}$, where $|T| = n_i(s_{\max} - s + 1)$, until $N(s)$ configuration samples for each task $T_{ij} \in T$ are obtained.
6: run successive halving (SH) on each task T_{ij} in bracket s, i.e. on $T^s = \{T_{ij} : i = 1, \ldots, n_i, j = s\}$
7: **end for**
8: **Return** Configuration with the optimal objective value for each task t_i.

† LCM(p, n) means an LCM model with p LCM-tasks, with n target samples for each LCM-task.

Relevant GPTune parameters. budget_min lowest fidelity $b_{\min}$. budget_max highest fidelity $b_{\max}$. budget_base the base η. See Section 6.1.10 for details.

3.1.8 Surrogate-based sensitivity analysis

Another use case of surrogate models is conducting sensitivity analysis[10] to determine how different values of individual tuning parameters could affect the output results. GPTune currently offers an interface for the sensitivity analysis tool based on the Sobol method [39] and using the implementation of a Python module called SALib [19]. The Sobol method is a global sensitivity analysis providing an assessment of parameter sensitivity. The Sobol analysis requires (1) samples drawn from the function directly, (2) evaluating the model using the generated sample inputs and saving the model output, and (3) conducting a variance-based mathematical analysis to compute the sensitivity indices. In the GPTune interface, we use a pre-trained surrogate model to draw and evaluate samples (we currently use the Saltelli sampling method [35, 36] from SALib [19])

Sobol analysis evaluates the part of the total variance of the response that can be attributed to the input parameter X_n. In the Sobol method, we usually consider two measures:

- First-order (Main-effect) indices: Linear (i.e., first-order) contribution (without interaction) of a parameter to the response variance.
- Total effect index: Total contribution (including higher order interactions among parameters) of a parameter to the response variance.

[10]This sensitivity analysis produces valid results for single-objective task only. For multi-objective task, it only produces marginal sensitivity and may produce misleading results, hence not recommended at the current stage.

A first-order (main-effect) index $S1_i$ is the contribution to the output variance of the main (linear) effect of an input attribute X_i, therefore, it measures the effect of varying X_i alone, but averaged over variations in other input parameters. It is standardized by total variance to provide a fractional contribution, its mathematical details can be found in [39, 36].

$$S1_i = \frac{Var_{X_i}\left(E_{\mathbf{X}_{\sim i}}(Y \mid X_i)\right)}{Var(Y)} \tag{15}$$

A total effect index ST_i represents the total contribution (including interactions) of a parameter X_i to the response variance. The index value is obtained by summing all first-order and higher-order effects involving the parameter X_i. We use the notation $\mathbf{X}_{\sim i}$ to denote conditioning on all attributes except X_i, and ST_i is therefore computed as follows:

$$ST_i = \frac{E_{\mathbf{X}_{\sim i}}\left(\mathrm{Var}_{X_i}(Y \mid \mathbf{X}_{\sim i})\right)}{\mathrm{Var}(Y)} = 1 - \frac{\mathrm{Var}_{\mathbf{X}_{\sim i}}\left(E_{X_i}(Y \mid \mathbf{X}_{\sim i})\right)}{\mathrm{Var}(Y)} \tag{16}$$

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 [7] and will be explained in detail. Fig. 2 illustrates the three supported modes regarding distributed-memory parallelism, i.e., the MPI spawning mode (the default), the Reverse Communication Interface (RCI) mode, and the lite mode.

1. **Default mode (MPI spawning mode)** . The OpenMPI>= 4.0.1 is a required dependency in this mode. For the default mode, the (MPI) application code needs to be compiled using the same software dependence as GPTune (OpenMPI, ScaLAPACK, etc.), and with the insertion of a few extra lines (see Section 6.1.1). This makes the application code callable from within the GPTune Python modules using the MPI spawning feature (see Figure 2 left). Note that the distributed-memory parallelism in the model and search phases is also supported via MPI spawning. In default mode, users of GPTune will need to provide the objective function (see the definition in Section 6.1.2) 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. More details and numerical experiments using the MPI spawning mode can be found in [26].

2. **RCI mode.** There are a few potential pitfalls of the default mode (spawning mode): 1. OpenMPI may not be available/installable on the user's machine. 2. The overhead of each MPI spawning can be quite high on some HPC platforms, and OpenMPI can give much worse communication performance compared to other MPI vendors such as SpectrumMPI, CrayMPICH and Intel MPI. 3. The default mode cannot handle application code failure well, i.e., checkpoint cannot be supported (one needs to manually restart the tuning once the application call fails, but the previous tuning data can still be reused). 4. The spawning mode requires that the MPI application code and GPTune are installed with the same OpenMPI version. To address these potential issues, the RCI mode does not require OpenMPI-compiled

application code or OpenMPI-compiled GPTune package, but requires a bash script to invoke GPTune and the application More specifically, the bash script will query GPTune (a Python driver) to find the next sample point and store it in GPTune's local database, retrieve the next sample point from the database, invoke the application code, write the evaluation results into the database, and then call the GPTune Python driver again to ask for the next sample (see Figure 2 right). As such, the parallelism of the objective function evaluation is completely handled in the bash script. Note that the RCI mode does not support distributed-memory parallelism in either the modeling or the search phases. The RCI mode will be explained in more detail in Sections 6.5.3 and 4.3.

3. **Lite mode.** If the user installs the lite version of GPTune, then it does not rely on OpenMPI or mpi4py. Compared to the RCI mode, the lite mode is more similar to the default mode (see Figure 2 middle), except that no distributed-memory parallelism is supported in the modeling, search and batched execution of the objective functions. Also, one cannot use the MPI spawning scheme to launch an MPI-based function evaluation. Instead, one has to either use "subprocess.run" to launch the objective function (see Section 3.2.5), or invoke a bash command inside Python, which launches the objective function. Both approaches can have significantly large overheads for running the MPI application code.

In the following Sections 3.2.1, 3.2.2, 3.2.3 and 3.2.4, the distributed-memory related contents only apply to the default (MPI spawning) mode, while the shared-memory related contents apply to all three modes. Distributed parallelism for the lite mode is described in Section 3.2.5.

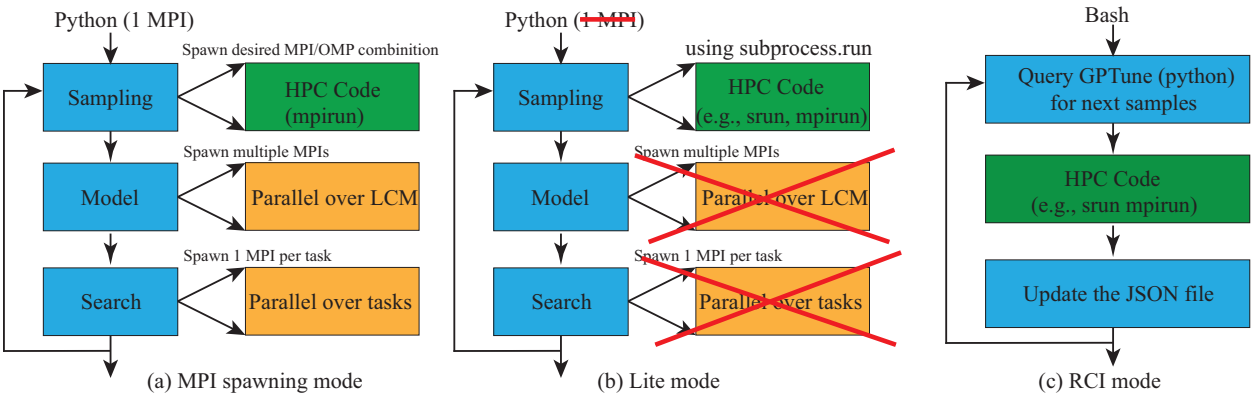

Figure 2: Three execution modes of GPTune: (a) MPI spawning-based interface (the default, full version of GPTune), (b) the lite mode, and (c) RCI mode.

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. 3.

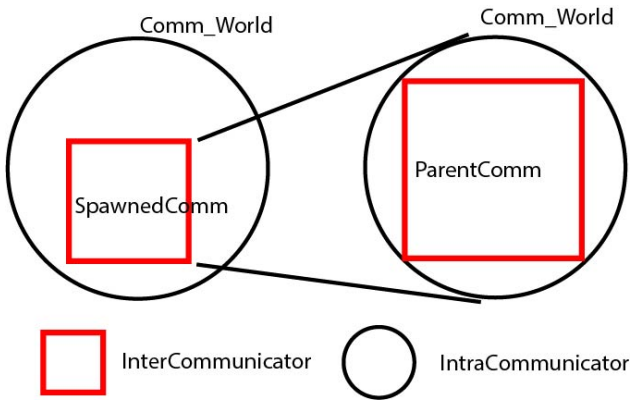

Figure 3: GPTune parallel programming model.

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 for 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 code 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 a future work.

3.2.2 Objective function evaluation

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 lines (“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 4.1) or communicate with the inter-communicators (see the example of SuperLU_Dist in Section 4.2). For example, assuming that 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 6.1.1) in the following fashion:

```
def objectives(point):
# extract task and tuning parameters, and global constants from "point", and use them to define environment 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 worker 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 variables, command lines and/or input files, compute the objective function, and write 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 for 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, and global constants from "point", and use them to define environment 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 worker 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 variables, 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 is beneficial to even perform multiple function evaluations simultaneously. GPTune uses MPI spawning and ThreadPoolExecutor to support distributed- and shared-memory parallelism over a number of function evaluations. Note that 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, for example, assigning each function evaluation a unique file or directory name.

Relevant GPTune parameters. objective_nprocmax: maximum core counts for each function evaluation. objective_evaluation_parallelism: whether to use parallelism on 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 thread processes for parallelizing over the number of function evaluations. See Section 6.1.10 for details. objective_nospawn: whether the application code is launched via MPI spawn. If True, options['objective_nprocmax'] cores are used per function evaluation, otherwise options['objective_nprocmax']+1 cores are used.

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 minimize the log-likelihood function using selected objective function samples. The GPTune implementation can choose n_{start} random starting guesses of the hyperparameters, each of which is used by L-BFGS to find 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 random starts n_{start} and the corresponding L-BFGS optimization are distributed over a user-specified number of threads or MPI processes. Note that currently one cannot use both shared-memory parallelism and distributed-memory parallelism (over n_{start}) at the same time. GPTune uses MPI spawning to support distributed-memory parallelism for 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.

Relevant 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 6.1.10 for more details.

3.2.4 Search phase of MLA

The search phase uses algorithms in PyGMO, pymoo or SciPy 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 n_i tasks using user-specified number of threads or MPI processes. Note that one cannot leverage both the shared-memory parallelism and distributed-memory parallelism (over n_i) at the same time. GPTune uses MPI spawning to support distributed-memory parallelism over the tasks. (2) For each task, the evolutionary algorithm (if PyGMO used) can be parallelized using user-specified number of threads.

Relevant GPTune parameters. distributed_memory_parallelism: whether to use distributed-memory parallelism over the n_i tasks. search_multitask_processes: number of MPI processes for parallelizing over n_i. shared_memory_parallelism: whether to use shared-memory parallelism over n_i. search_multitask_threads: number of threads for parallelizing over n_i. search_threads: number of threads used in PyGMO. See Section 6.1.10 for more details.

3.2.5 Objective function evaluation (lite mode)

If the lite version of GPTune is installed, one can still launch the MPI-compiled application code from inside Python, using e.g., the subprocess module, instead of the MPI spawning from the mpi4py module. Note that since MPI spawning is not used, one typically does not need to modify the application code.

The subprocess module provides utility functions to set the environment variables, launch the MPI-compiled code, and extract objective function values from the runlogs (or stdout). For example, environment variables can be set by

```
my_env = os.environ.copy()
my_env["OMP_NUM_THREADS"]=str(nthreads)
```

where "nthreads" is the number of threads possibly contained in the task or tuning parameters. The application can be launched with command line arguments using

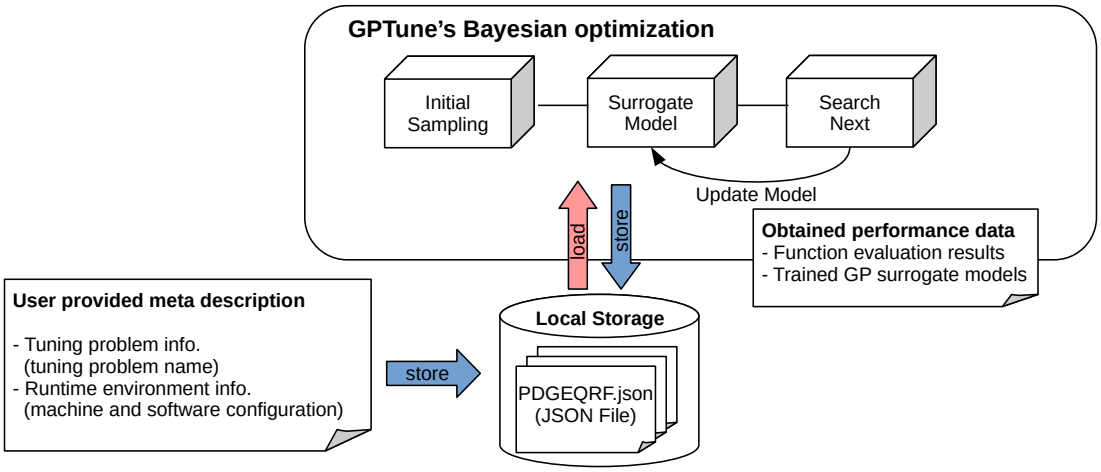

Figure 4: Design of the GPTune database.

```
args = ['-a1', '%s'%(av1), '-a2', '%s'%(av2)] # command line arguments
mpirun_command = os.getenv("MPIRUN") # $GPTUNEROOT/run_env.sh will set MPIRUN to
    mpirun/srun/jsrun, etc.
argslist = [mpirun_command,'-n',str(nproc),'path-to-executable',args] # nproc is
    the number of MPIs launching the application
p = subprocess.run(argslist,capture_output=True,env=my_env) # capture_output will
    enable recording of stdout and stderr from the application run
```

Depending on how the application code is implemented, parameters can be passed through command lines ("args"), environment variables ("my_env"), or an input file stored on disc.

To collect the returned value(s) from the application, here we assume that the value can be extracted from reading the runlog.

```
stdout = p.stdout.decode('ascii')
print(stdout) # this can be commented out if one wants to hide the logs
resultline = stdout.split("\n") # parse the runlog by lines
read the objective values from resultline
```

A complete example of using subprocess to launch applications can be found at `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST/superlu_MLA.py`. To use this example, it is assumed that one has installed SuperLU and its dependencies following Section 2.2.9. Note that one needs to set the environment variable

```
export GPTUNE_LITE_MODE=1
```

to activate subprocess-based application launching.

3.3 Database support

3.3.1 Design

GPTune automatically stores and loads historical performance data obtained from autotuning (e.g., function evaluations, trained surrogate models) to and from performance data files with JavaScript Object Notation (JSON) [31] format in the user's local storage. Database support

facilitates many useful autotuning features such as error handling, checkpointing, benchmarking, and transfer learning. Figure 4 illustrates how the history database runs in conjunction with GPTune's Bayesian optimization.

Each application (or each tuning problem) has a separate JSON data file that contains all the historical function evaluation results of the tuning problem. After evaluating each parameter configuration, GPTune stores the task parameters, the tuning parameters, and the result evaluated in the JSON file. This practice ensures that no data are lost in the cases where (a) a long tuning experiment with very expensive function evaluations does not complete due to time limitation, or (b) certain parameter configurations crash and cause the tuner to stop. If GPTune is run in parallel and multiple processes attempt to update the JSON file simultaneously, the history database allows only one process to update the file at a time based on simple file access control (either using Python's `filelock` module or using `rsync`-based file synchronization). If the user wants to reset the tuning database for certain purposes, the user can simply remove/move the database file.

GPTune also supports storing and loading trained GP surrogate models along with some model statistics information, such as likelihood values of the model. Users can use this feature to not only reuse pre-trained models for autotuning, but also use the model for sensitivity analysis.

GPTune can also record the runtime environment information such as machine configuration and software information (i.e., which software libraries and versions are used for that application) along with the function evaluation results, via user-provided meta descriptions. Then, if the user uses the database for checkpointing and restarting of the autotuner, GPTune can ensure that it loads function evaluations collected from the same runtime environment. In addition, for TLA, users can determine which data are relevant for learning from possibly different machines or software configurations. In Section 6.3, we explain the user interfaces for querying relevant performance data based on runtime environments.

3.3.2 Meta description

In a meta description, users can provide the name of the tuning problem (which is the name of the database file, e.g., PDGEQRF.json), compute resources needed, and software dependencies for the given tuning experiment. Therefore, GPTune can record relevant tuning information and the environment information in the database. Users can use either a Python dictionary or a meta description JSON file to provide such information, for example:

Using a Python Dictionary. Listing 3.3.2 shows an example meta description using Python Dict data type.

```
tuning_metadata = {
    "tuning_problem_name": "PDGEQRF",
    "machine_configuration": {
        "machine_name": "Cori",
        "haswell": {
            "nodes": 1,
            "cores": 32
        }
    },
    "software_configuration": {
        "openmpi": {
            "version_split": [4,0,1]
        },
```

```
        "scalapack": {
            "version_split": [2,1,0]
        },
        "gcc": {
            "version_split": [8,3,0]
        }
    }
}
```

Information related to machine configuration includes the machine name (e.g., Cori) and the number of cores/nodes used. The software configuration includes the dependent software packages that are used to compile and install the application as well as their version information.

The meta description must be defined in the GPTune Python driver, and the dictionary `tuning_metadata` must be linked when the user creates a database instance (Section 6.1.5).

If the user does not need to record and maintain the runtime environment (e.g., if the user is tuning a mathematical function, where the objective function is not affected by any computational resources such as CPU and memory, the user can skip providing `machine_configurations` and `software_configurations`.

Users can read some useful information from the meta description, using the following wrapper function.

```
(machine, processor, nodes, cores) = GetMachineConfiguration(meta_dict =
    tuning_metadata)
# read meta-information defined in the tuning_metadata
```

Using a meta description file. The meta description file must be in JSON format. In case the user wants to use a meta description file, the default path to the meta description file is located at ./.gptune/meta.json. While the meta description file can be edited manually, the meta file can be automatically generated using jq from the command line. For example:

```
tp=application_name      # define application name
nodes=1                        # define number of compute nodes
cores=4                       # define number of cores per node
machine=mymachine      # define machine name
proc=intel-i7                       # define processor type

# generate current software information. Change the software names and versions as
    needed.
software_json=$(echo ",\"software_configuration\":{\"openmpi\":{\"version_split\":
    [4,0,1]},\"scalapack\":{\"version_split\": [2,1,0]},\"gcc\":{\"version_split\":
    [8,3,0]}}")
# generate current machine information
machine_json=$(echo ",\"machine_configuration\":{\"machine_name\":\"$machine\",\"$
    proc\":{\"nodes\":$nodes,\"cores\":$cores}}")
# generate the meta file with jq
app_json=$(echo "{\"tuning_problem_name\":\"$tp\"")
mkdir -p .gptune
echo "$app_json$machine_json$software_json}" | jq '.' > .gptune/meta.json
```

In case a meta description file is used, users can read some useful information from the meta description, using the following wrapper function.

```
(machine, processor, nodes, cores) = GetMachineConfiguration(meta_path=./gptune/
    meta.json)
```

```
# read meta information defined in the ./gptune/meta.json file
```

The user can specify the meta description file path using the `meta_path` argument. If none, the function will try to find the information from ./gptune/meta.json.

3.3.3 Automatic environment parsing

The machine and software configuration in the meta description can be provided automatically with our automatic environment parsing, in case the user is using some popular HPC software tools such as Slurm [38] and Spack [13]. Below, using an example of PLASMA's DGEQRF tuning, we show how the user can use automatic parsing and simplify the meta description.

- Manually given runtime environment information.

```
tuning_metadata = {
    "tuning_problem_name": "DGEQRF",
    "tuning_problem_category": "PLASMA",
    "machine_configuration": {
        "machine_name": "Cori",
        "haswell": { "nodes": 1, "cores": 32 }
    },
    "software_configuration": {
        "openblas": {
            "version_split":[0,3,17]
        },
        "plasma":{
            "version_split":[20,9,20]
        }
    }
}
```

- Parsing software configuration from Spack/CK. This example shows that the user can replace the `software_configuration` with a simple sentence `"spack":["plasma"]` which means that the PLASMA software was installed and loaded using Spack. GPTune will then automatically record the version of the PLASMA library along with the versions of the dependent software packages.

```
tuning_metadata = {
    "tuning_problem_name": "DGEQRF",
    "tuning_problem_category": "PLASMA",
    "machine_configuration": {
        "machine_name": "Cori",
        "haswell": { "nodes": 1, "cores": 32 }
    },
    "spack": ["plasma"]
}
```

- Parsing machine configuration from Slurm. Slurm is one of the most popular job management systems for HPC systems. We support parsing the machine configuration if the user is running the tuning process based on a Slurm environment. The user can replace `machine_configuration` with a simple pragma `"slurm":"yes"`.

```
tuning_metadata = {
    "tuning_problem_name": "DGEQRF",
    "tuning_problem_category": "PLASMA",
    "slurm": "yes",
    "spack": ["plasma"]
}
```

3.3.4 JSON data format.

Here, we describe the JSON data format for storing GPTune performance data. Each JSON database file has two labels `func_eval` and `model_data`. As the name indicates, `func_eval` contains the list of all function evaluation results, and `surrogate_model` contains the list of the meta-data for each trained surrogate model.

```
{
  "func_eval": [
    {
      /* function evaluation result */
    },
    {
      /* function evaluation result */
    },
    ...
  ],
  "surrogate_model": [
    {
      /* surroagte model meta-data */
    },
    {
      /* surroagte model meta-data */
    },

    ...
  ]
}
```

Function Evaluations. Listing 1 shows a function evaluation result of the *PDGEQRF* routine of ScaLAPACK [3] for a given task/parameter configuration. In the listing, `task_parameter` contains the information about the task parameter, and `tuning_parameter` contains the tuning parameter configuration, and its evaluation result is stored in `evaluation_result`. Machine and software configurations provided by the user-given meta description are stored in `machine_configuration` and `software_configuration`, respectively. Also, when saving a function evaluation result, the data-creation time and a unique ID (UID) of the function evaluation are automatically generated and appended by GPTune. If different users submit function evaluation results for the same task and parameter configurations, the database can differentiate between different function evaluation results based on their UIDs.

Please refer to `https://gptune.lbl.gov/repo/export-all/?tuning_problem_name=PDGEQRF&search=["func_eval"]` for some example function evaluation records of ScaLAPACK's PDGEQRF tuning problem.

```
{
```

```
  "task_parameter": {
    "m": 30000,
    "n": 30000
  },
  "tuning_parameter": {
    "mb": 5,
    "nb": 13,
    "npernode": 2,
    "p": 9
  },
  "evaluation_result": {
    "r": 15.148637
  },
  "machine_configuration": {
    "machine": "cori",
    "haswell": {
      "nodes": 8,
      "cores": 32
    },
    "knl": {
      "nodes": 0,
      "cores": 0
    }
  },
  "software_configuration": {
    "openmpi": {
      "version_str": "4.0.0",
      "version_split": [
        4,
        0,
        0
      ],
      "tags": "lib,mpi,openmpi"
    },
    "scalapack": {
      "version_str": "2.1.0",
      "version_split": [
        2,
        1,
        0
      ],
      "tags": "lib,scalapack"
    }
  },
  "time": {
    "tm_year": 2021,
    "tm_mon": 1,
    "tm_mday": 27,
    "tm_hour": 21,
    "tm_min": 22,
    "tm_sec": 22,
    "tm_wday": 2,
    "tm_yday": 27,
    "tm_isdst": 0
  },
  "uid": "cc7c03ec-6128-11eb-a40f-85c4081a47e2"
```

```
}
```

Listing 1: Example Function Evaluation Result

Surrogate Models. GPTune supports several different surrogate modeling options, and the GPTune history database can record the meta information of the trained surrogate model. The meta information can be loaded to reproduce the same surrogate model, without needing to re-train the model.

- Parallel-version LCM modeling (`model_class=Model_LCM`): Here, we assume GPTune's distributed parallelism-enabled multi-task modeling scheme (Section 3.1), based on the LCM, is used (the full version of GPTune is needed). Listing 2 shows the information of a surrogate model for the IJ routine of Hypre [10] for five different function evaluation results for task {i: 200, j: 200, k: 200}.
  - `hyperparameters` contains the hyperparameter values which are required to reproduce the surrogate model. Recall that in LCM, equations (3) and (4), the hyperparameters are l_j^q, $a_{i,q}$, σ_q, $b_{i,q}$, d_i. Hence, in the following example which considers one task ($Q = 1$) for 12 tuning parameters ($\beta = 12$), we store 16 hyperparameters in total (12 for l_j^q and 1 for each of $a_{i,q}$, σ_q, $b_{i,q}$, d_i). As another example, considering two tasks ($Q = 2$) for 12 tuning parameters ($\beta = 12$), we need to store 36 hyperparameters in total (24 for l_j^q, 4 for $a_{i,q}$, 2 for σ_q, 4 for $b_{i,q}$, 2 for d_i).
  - `model_stats` stores the model's statistics information. For the GPTune's LCM, we can store some statistics information such as *log_likelihood*, *neg_log_likelihood*, *gradients*, and *iteration* (how many iterations were required to converge the model). Note that stored information of the trained surrogate models may or may not be meaningful for different problem spaces.
  - The JSON data also contain information about task parameters (`task_parameters`) and which function evaluation results were used (`func_eval`) to build the surrogate model, by containing the list of the UIDs of the function evaluation results.

  The history database can load trained models only if they match the problem space of the given optimization problem. Similar to the function evaluation results, the data generation time and a unique ID of each surrogate model are also automatically appended by GPTune.
- Standard GP model using GPy (`model_class=Model_GPy_LCM`): This section assumes the GPy package-based modeling is used. If the user is using the lite version of GPTune or if the user is using a single-node computer, the user might use this modeling option. We also support recording the hyperparameter information of the GP model, while also considering the variants of the GP kernel selection such as RBF (Matern ∞), Matern12, Matern32, Matern52 (See [33] for detail definitions of kernels).

Listing 2 in below shows a simplified example of surrogate model data for the PDGEQRF example (using the `Model_LCM` option). Please refer to `https://gptune.lbl.gov/repo/export-all/?tuning_problem_name=PDGEQRF&search=["surrogate_model"]` for the detailed example record(s) of ScaLAPACK's PDGEQRF tuning problem.

```
{
  'accessibility': {'type': 'public'},
  'document_type': 'surrogate_model',
  'function_evaluations': ['7eeb110e-2e81-11ed-8e73-05578899d507',
                           '81d47798-2e81-11ed-8e73-05578899d507',
                           ...
                           '66c778aa-2e82-11ed-8e73-05578899d507'],
  'hyperparameters': [41.93264006653109,
                      0.037043444858459534,
                      0.34979125064736544,
                      4.5268584980621585,
                      1.0,
                      0.0258010876780688,
                      4.588786428201813e-06,
                      0.030303443634493256],
   'model_stats': {'gradients': [4.1487544200647726e-08,
                                 ...
                                 4.022373047973815e-07],
                   'iterations': 27,
                   'log_likelihood': 233.57298522977464,
                   'neg_log_likelihood': -233.57298522977464},
  'input_space': [{'lower_bound': 128,
                   'name': 'm',
                   'optimize': True,
                   'transformer': 'normalize',
                   'type': 'int',
                   'upper_bound': 1000},
                  {'lower_bound': 128,
                   'name': 'n',
                   'optimize': True,
                   'transformer': 'normalize',
                   'type': 'int',
                   'upper_bound': 1000}],
  'modeler': 'Model_LCM',
  'objective': {'lower_bound': -inf,
                'name': 'r',
                'objective_id': 0,
                'optimize': True,
                'transformer': 'identity',
                'type': 'real',
                'upper_bound': inf},
  'output_space': [{'lower_bound': -inf,
                    'name': 'r',
                    'optimize': True,
                    'transformer': 'identity',
                    'type': 'real',
                    'upper_bound': inf}],
  'parameter_space': [{'lower_bound': 1,
                       'name': 'mb',
                       'optimize': True,
                       'transformer': 'normalize',
                       'type': 'int',
                       'upper_bound': 16},
                      {'lower_bound': 1,
                       'name': 'nb',
                       'optimize': True,
```

```
                        'transformer': 'normalize',
                        'type': 'int',
                        'upper_bound': 16},
                       {'lower_bound': 0,
                        'name': 'lg2npernode',
                        'optimize': True,
                        'transformer': 'normalize',
                        'type': 'int',
                        'upper_bound': 5},
                       {'lower_bound': 1,
                        'name': 'p',
                        'optimize': True,
                        'transformer': 'normalize',
                        'type': 'int',
                        'upper_bound': 32}],
  'task_parameters': [[1000, 1000]],
  'time': {'tm_hour': 0,
           'tm_isdst': 1,
           'tm_mday': 7,
           'tm_min': 55,
           'tm_mon': 9,
           'tm_sec': 15,
           'tm_wday': 2,
           'tm_yday': 250,
           'tm_year': 2022},
  'tuning_problem_name': 'PDGEQRF',
  'uid': '68a6548e-2e82-11ed-8e73-05578899d507'
}

```

Listing 2: Example surrogate model (simplified).

3.4 Crowd-based autotuning

It is common that, for a popular application, multiple users will need to run different tuning instances possibly on different HPC systems or different problem sizes. To take advantage of all the available historical data from various users, we provide a public shared repository at `https://gptune.lbl.gov` through NERSC's Science Gateways [30], where users can upload their performance data obtained from GPTune and download performance data provided by other users. In what follows, we provide an overview of the shared database support which harnesses the crowd for autotuning. For more information about the shared database, we also have an online-version manual at `https://gptune.lbl.gov/docs`.

3.4.1 Shared database on the cloud

Overview. Figure 5 shows the design and the use cases of the shared database with an illustration of how the shared database handles users' database queries. In the shared database, all collected performance data (e.g., function evaluations and trained surrogate models) is stored in storage provided by NERSC and managed via MongoDB [6], so it is easy to convert collected data into a JSON format. In the shared database, each performance data record uses the same data format as GPTune's local database, discussed in Section 3.3.4. Therefore, GPTune users can submit the obtained database file (or obtained function evaluations) to the shared database without any

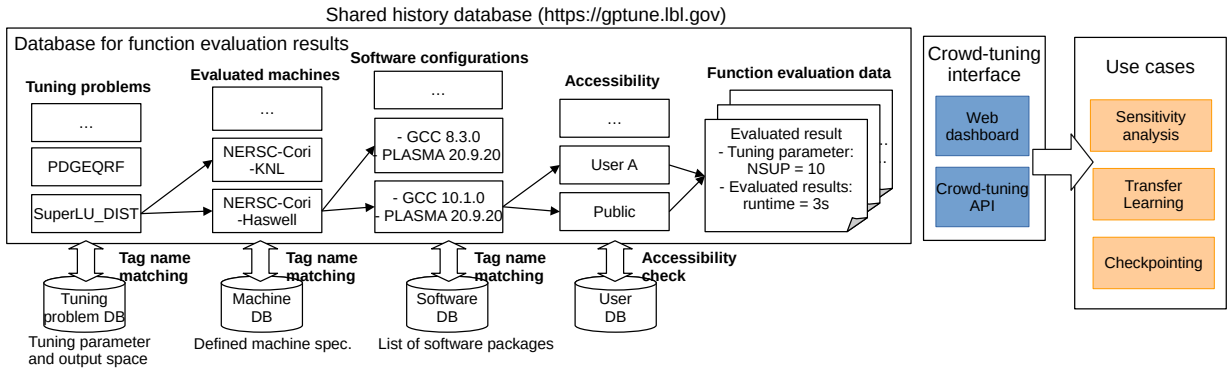

Figure 5: Design of the shared database.

modifications and reuse performance data provided by other users. Regardless of that, any other autotuner users can also use the shared database, as long as they submit data records respecting our JSON format.

Reliability and security. Interested users can sign-up at `https://gptune.lbl.gov/account/signup/`. For reliability, the shared database allows only registered users to upload performance data and stores the user information (i.e., username and email) of each performance data record. However, we also provide an option for users who do not wish to disclose their information to other users; in that case, we anonymize the submitted performance data. When uploading performance data, the user can specify a certain accessibility level: private, shared by registered users or specific user groups, or publicly available. To maintain security, all requests to the repository, either using the programmable API or the interactive dashboard, have to be done over HTTPs. In addition, we have applied a number of techniques (e.g., robot checking, user password protection, and regular vulnerability checking) to assure the security of the database.

Reproducibility. As discussed in Section 3.3.2, GPTune records the runtime environment of each function evaluation result. The shared database also records the runtime environment and that information can be used for users to query relevant performance data from the shared database. One challenge is that, since the runtime environment can be given manually, different users might use different names to describe the machine and software configuration. To address this challenge, the shared repository can parse the user provided information internally to match the tag names with the well-defined machine software information in the database. As shown in Figure 5, we maintain separate databases for the detailed information of popular software frameworks and user systems with possible tag names for this purpose. Users can also add their own machine information to the database, or request to add more data into our software and machine databases.

Usability and programmability. Finally, users can use this shared database either using a web-based interface or a programmable interface which is called the *crowd-tuning API*, to access the database and query relevant performance data. These interfaces are used for harnessing the crowd for autotuning via GPTune's TLA and/or sensitivity analysis. Using the interface, end-users can even query the best available tuning parameter to run a certain HPC code, without needing to perform any autotuning.

3.4.2 Web dashboard

This section describes the web-based interface, where users can access the database using a web browser and download/upload performance data.

Downloading performance data. To download tuning data (e.g., function evaluation results, surrogate models) from the shared database, we provide an interactive dashboard at `https://gptune.lbl.gov/repo/dashboard/`. On the dashboard, users first need to select a tuning problem from the drop-down menu. The drop-down menu will show the names of available tuning problems along with their categories. Once a tuning problem is selected, the dashboard will display all available machine configurations, software configurations, and the owner information. The user can check the machine/software/user configuration(s) that match the user's interests. Then, the dashboard will display a table that contains all the filtered results and allow users to download in various formats, including JSON and CSV (Comma Separated Values). Note that the dashboard displays results based on the user's privilege level. If the user is using the dashboard without signing in, the user will only be able to access publicly available data.

Uploading performance data. To upload function evaluation results and/or surrogate model data, the user can use an online form at `https://gptune.lbl.gov/repo/upload/`. The user first needs to select the tuning problem. If the user's tuning problem is not shown in the drop-down menu, the user needs to define (add) the tuning problem in the shared database (see the paragraph below about how to add a tuning problem). Once the tuning problem and the machine are selected, the user can upload data either by using a JSON data file (generated by GPTune) or by using text data in the JSON format.

Define tuning problems. Before uploading any performance data, users need to define their tuning problems in the shared repository unless the same tuning problem already exists in the repository. Based on the (well-defined) tuning problem information, multiple users will be able to run autotuning for the same tuning problem and share performance data.

The list of available tuning problems is shown at `https://gptune.lbl.gov/repo/tuning-problems/`. To add a new tuning problem, the user can use an online form at `https://gptune.lbl.gov/repo/add-tuning-problem/`. The user needs to define the tuning name and select appropriate category/categories of the tuning problem. Then, the user needs to define the task parameters, the tuning parameter parameters, and the output parameters, with required constant variables (if there are any).

Adding machine and software information. As discussed in Section 3.4.1, our shared database performs internal tag name matching, because different users can use (slightly) different names to refer to the same machine or software information. We have built and maintain separate databases that contain information about popular hardware architectures and software packages, but users are also welcome to provide more information to enrich our database.

Users can define a new machine using an online form at `https://gptune.lbl.gov/repo/add-machine/`. The user first needs to provide the machine name and the site/institute information. The user can then select the system model type (e.g., HPC systems manufacturer, cloud service provider, etc.). Another important data field is to provide the processor types of the machine. We have built a tree-based processor list from popular (HPC) processor vendors such as Intel,

NVIDIA, IBM, AMD, ARM, etc. The user can select one processor type (homogeneous system) or multiple processor types (heterogeneous system) that make up the user's machine and provide information about the number of nodes/cores contained in the machine. The user can finally select the interconnect(s) of the machine and submit the form. The currently available machine information list is shown at `https://gptune.lbl.gov/repo/machines/`. For the software database, we have obtained a tree-based structure of widely used software packages/tools, from CK's software database. We believe that the provided list can cover many different software settings, but users can contact us at gptune-dev@lbl.gov if there are missing software packages/tools/datasets.

3.4.3 Crowd-tuning API

The crowd-tuning API is a programmable interface to access the repository via HTTPs based on REpresentational State Transfer (REST) principles, which means that the database resources can be identified via a Uniform Resource Identifier (URI) and accessed in various programming languages like C++/Python.

API key. Unlike the web-dashboard where the user can login from a web browser, all API requests need to contain an API key with a x-api-key header (e.g., `"X-Api-Key":"your_API_key"`). Each user can generate one or more API keys at `https://gptune.lbl.gov/account/access-tokens` with an option to choose an expiration date. When generating an API key, the user can select an option whether the user agrees to disclose the user info or prefers to be anonymous. To enhance the security of API keys, we provide an additional option to use public and private key pairs for API keys. In this case, the user needs to keep the private key and we record only the public key in our user database. Note that, users have to manage their API keys securely, because API keys are used instead of passwords.

Generalized API format. To download performance data, the user can use a general URL form in the following, where `tuning_problem_name` is the name of tuning problem, and `machine_configurations`, `software_configurations` and `user_configurations` contain lists of machine, software, and user configurations to download.

```
GET https://gptune.lbl.gov/direct-download/?tuning_problem_name=ScaLAPACK-PDGEQRF&
    machine_configurations=[ ... ]&software_configurations=[ ... ]&
    user_configurations=[ ... ] }
```

Similarly, the general URL form for uploading a function evaluation result is as follows, where `tuning_problem_name` is the name of tuning problem to be queried, and `function_evaluation` is the function evaluation result to be uploaded.

```
POST https://gptune.lbl.gov/direct-update/?tuning_problem_name=ScaLAPACK-PDGEQRF&
    function_evaluation={ ... }
```

The function evaluation data contains the tuning parameter configuration and its evaluated outputs, and the machine and software configuration on which the parameter configuration is evaluated, following the JSON format in Section 3.3.4.

Accessing the shared database in Python. As an example, we explain how to use this API from a Python script using Python's `requests` module. Using this feature, users can use their own scripts to query performance data in the repository.

To query performance data, the user can use the requests module's `get` function as follows. The user needs to provide an API key in the headers, and provide machine configurations and software configuration(s) that the user wants to allow downloading. The function will then return a dictionary value (e.g., `r` in the code below) that contains the downloaded performance data (e.g., `r["perf_data"]`) as well as the request response data (e.g., `"r.status_code==200"` indicates that the request was successful).

```
import requests
r = requests.get(url="https://gptune.lbl.gov/direct-download",
                 headers={"x-api-key":"your_api_key"},
                 params={"tuning_problem_name":"your_tuning_problem",
                         "machine_configurations": [
                             {"cori": {"haswell": {"nodes":1, "cores":32}}},
                             {"cori": {"knl": {"nodes":1, "cores":68}}}
                         ],
                         "software_configurations": [
                             {"gcc": {"version_split":[8,3,0]}}
                         ]
                 }
                 verify=False)
```

To upload performance data, the user can use the `requests` module's `post` function. The user needs to provide an API key in the headers and provide the function evaluation. The function returns a response message.

```
import requests
r = requests.post(url = "https://gptune.lbl.gov/direct-upload",
                  headers={"x-api-key":"your_api_key"},
                  data={"tuning_problem_name":"your_tuning_problem_name",
                        "function_evaluation_document": {
                            "task_parameter": { "m": 10, "n": 20 },
                            "tuning_parameter": { "mb": 2, "nb": 1 }
                            "evaluation_result": { "runtime": 6 }
                        }
                  }
                  verify=False)
```

Utility wrapper functions. Based on the programmable shared database access, we further provide a number of useful Python wrapper functions that can be used for various tuning scenarios. Currently we provide five wrapper functions that perform useful crowd-based autotuning techniques using performance data in the shared database, as follows:

- Querying performance data from the shared database.
- Upload performance data to the shared database.
- Query a surrogate performance model from the shared database.
- Make a output (performance) prediction for a given tuning parameter configuration.
- Run a sensitivity analysis for a given parameter space.

Those functions are available within the GPTune packages. The detailed user interfaces for these functions are given in Section 6.4.

Run GPTune with the cloud database using an API key . Users can provide an API key to run the GPTune autotuner using the HistoryDB repository. In other words, historical performance data can be downloaded automatically from the repository, and obtained function evaluation results can also be submitted to the repository automatically. To do so, the user only needs to modify the tuning meta description (Section 3.3.2) in the GPTune driver to enable the shared database access.

The following is an example meta description which enables access to the shared database.

```
tuning_metadata = {
    "tuning_problem_name": "PLASMA-DGEQRF",
    "historydb_api_key": "your_api_key",
    "use_crowd_repo": "yes",
    "slurm": "yes",
    "spack": ["plasma"]
}
```

The pragma `"use_crowd_repo":"yes"` is used to invoke the shared database feature, and the pragma `"historydb_api_key"="your_api_key"` is necessary to pass the API key.

4 Illustrative Examples

4.1 ScaLAPACK QR

Here, we use ScaLAPACK's *PDGEQRF* to illustrate code examples for using GPTune's tuning features. As mentioned in Section 1.2, we use (m, n) to define a task, then $(mb, nb, npernode, p)$ to define the tuning parameters. Note that we use simplified codes to help understand the code interface; please refer to the complete working codes at

- The GPTune Python driver: `https://github.com/gptune/GPTune/blob/master/examples/Scalapack-PDGEQRF/scalapack_MLA.py`
- The Python wrapper to invoke the ScaLAPACK application `https://github.com/gptune/GPTune/blob/master/examples/Scalapack-PDGEQRF/scalapack-driver/spt/pdqrdriver.py`
- The Fortran driver from the application side. `https://github.com/gptune/GPTune/blob/master/examples/Scalapack-PDGEQRF/scalapack-driver/src/pdqrdriver.f`
- The bash script to run the tests: `https://github.com/gptune/GPTune/blob/master/examples/Scalapack-PDGEQRF/run_examples.sh`

4.1.1 MLA

This example builds an LCM model of the *PDGEQRF* example for two user-specified tasks [[400,500], [800,600]]. The Python interface (pdqrdriver.py) to the Fortran application (pdqrdriver.f) will write the task parameter and tuning parameters into an input file named "GPTune/examples/Scalapack-PDGEQRF/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-PDGEQRF/scalapack-driver/exp/MACHINE_NAME/TUNER_NAME/JOBID/QR.out". The variables "MACHINE_NAME", "TUNER_NAME", and "JOBID" can be defined by the user. As discussed in Section 3.3, the code defines a meta description of the

tuning problem in in a Python dictionary named "tuning_metadata", so that GPTune's database can record and maintain the obtained function evaluation results with the environment information, machine and software configuration.

In terms of computation resources, "nodes=1" and "cores=16" are used for GPTune's several phases and for invoking the application code (pdqrdriver.f). As we use MPI spawn to invoke the application, one process is reserved as the spawning process. Therefore, it is recommended allocating "nodes+1" compute nodes with 1 node reserved for MPI spawning.

Note that to reduce the runtime noise, the Python driver will execute the same task and tuning parameter configuration three times (niter=3 as an argument of pdqrdriver) 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%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):
    # global constants defined in TuningProblem
    nodes = point['nodes']
    cores = point['cores']
    bunit = point['bunit']
    # task and tuning parameters
    m = point['m']
    n = point['n']
    mb = point['mb']*bunit
    nb = point['nb']*bunit
    p = point['p']
    npernode = 2**point['lg2npernode']
    nproc = nodes*npernode
    nthreads = int(cores / npernode)
    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
    q = int(nproc / p)
    nproc = p*q
    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 JOBID
    mmax = 2000  # maximum row dimension
    nmax = 2000  # maximum column dimension
    ntask = 2   # 2 tasks used for MLA
    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

    # defining tuning metadata
    tuning_metadata = {
        "tuning_problem_name": "PDGEQRF",
        "machine_configuration": {
            "machine_name": "Cori",
            "haswell": {
                "nodes": 1,
                "cores": 32
            }
        },
        "software_configuration": {
            "openmpi": {
                "version_split": [4,0,1]
            },
            "scalapack": {
                "version_split": [2,1,0]
            },
            "gcc": {
                "version_split": [8,3,0]
            }
        },
        "loadable_machine_configurations": {
            "Cori" : {
                "haswell": {
                    "nodes":1,
                    "cores":32
                }
            }
        },
        "loadable_software_configurations": {
            "openmpi": {
                "version_from":[4,0,1],
                "version_to":[5,0,0]
            },
            "scalapack":{
                "version_split":[2,1,0]
            },
            "gcc": {
                "version_split": [8,3,0]
            }
        }
    }

    (machine, processor, nodes, cores) = GetMachineConfiguration(meta_dict =
    tuning_metadata)
    print ("machine: " + machine + " processor: " + processor + " num_nodes: " +
    str(nodes) + " num_cores: " + str(cores))
    os.environ['MACHINE_NAME'] = machine
    os.environ['TUNER_NAME'] = TUNER_NAME

    """ 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 , 16, transform="normalize", name="mb")
    nb = Integer(1 , 16, transform="normalize", name="nb")
    lg2npernode = Integer(0, int(math.log2(cores)), transform="normalize", name="lg2npernode")
    p = Integer(1 , nodes*cores, transform="normalize", name="p")
    PS = Space([mb, nb, lg2npernode, p])
    # Output
    r = Real(float("-Inf") , float("Inf"), name="r")
    OS = Space([r])
    # Constraints
    cst1 = "mb*bunit*p<=m"
    cst2 = "nb*bunit*nodes*2**lg2npernode<=n*p"
    cst3 = "nodes*2**lg2npernode>=p"
    constraints = {"cst1" : cst1, "cst2" : cst2, "cst3" : cst3}
    constants={"nodes":nodes, "cores":cores, "bunit":bunit}
    print(IS, PS, OS, constraints)

    problem = TuningProblem(IS, PS, OS, objective, constraints, None, constants)
    historydb = HistoryDB(meta_dict=tuning_metadata)
    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, historydb = historydb)

    """ Building MLA with the given list of tasks """
    giventask = [[1000,1000],[500,500]]
    NI = len(giventask)
    (data, models,stats) = gptune.MLA(NS=NS, NI=NI, Tgiven =giventask, NS1 = max(NS//2,1))
    print("stats: ",stats)
    """ Print all 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].tolist())
        print('    Popt ', data.P[tid][np.argmin(data.O[tid])], 'Oopt ', min(data.O[tid])[0], 'nth ', np.argmin(data.O[tid]))

if __name__ == "__main__":
```

```
188     main()
```

Listing 3: scalapack QR example: scalapack_MLA.py.

```
1       PROGRAM PDQRDRIVER
2       CHARACTER*200       FILEDIR
3       INTEGER             IAM
4       INTEGER             master
5       PARAMETER           ( NIN = 1, NOUT = 2)
6       DATA                KTESTS, KPASS, KFAIL, KSKIP /4*0/
7
8 *     Get starting information
9       CALL GETARG(1,FILEDIR)
10      CALL MPI_INIT(ierr)
11      CALL MPI_COMM_GET_PARENT(master, ierr)
12      CALL BLACS_PINFO( IAM, NPROCS )
13
14 *     Open input file
15      OPEN( NIN, FILE=trim(FILEDIR)//'QR.in', STATUS='OLD' )
16      IF( IAM.EQ.0 ) THEN
17          OPEN( NOUT, FILE=trim(FILEDIR)//'QR.out', STATUS='UNKNOWN' )
18      END IF
19
20 *     Read number of configurations
21      READ( NIN, FMT = 1111 ) NBCONF
22 *     Print headings
23      IF( IAM.EQ.0 ) THEN
24          WRITE( NOUT, FMT = * )
25          WRITE( NOUT, FMT = 9995 )
26          WRITE( NOUT, FMT = 9994 )
27          WRITE( NOUT, FMT = * )
28      END IF
29
30 *     Run the computation for NBCONF times and dump the runtime to QR.out
31      DO 50 CONFIG = 1, NBCONF
32 *       ...
33 *       ...
34      END DO
35
36 *     Print out ending messages and close output file
37      IF( IAM.EQ.0 ) THEN
38         KTESTS = KPASS + KFAIL + KSKIP
39         WRITE( NOUT, FMT = * )
40         WRITE( NOUT, FMT = 9992 ) KTESTS
41         IF( CHECK ) THEN
42            WRITE( NOUT, FMT = 9991 ) KPASS
43            WRITE( NOUT, FMT = 9989 ) KFAIL
44         ELSE
45            WRITE( NOUT, FMT = 9990 ) KPASS
46         END IF
47         WRITE( NOUT, FMT = 9988 ) KSKIP
48         WRITE( NOUT, FMT = * )
49         WRITE( NOUT, FMT = * )
50         WRITE( NOUT, FMT = 9987 )
51      END IF
52
```

```
*     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 4: pdqrdriver.f.

In order to run the tuning experiment, we use the following mpirun command:

```
mpirun --oversubscribe --allow-run-as-root --mca pmix_server_max_wait 3600 --mca
    pmix_base_exchange_timeout 3600 --mca orte_abort_timeout 3600 --mca
    plm_rsh_no_tree_spawn true -n 1  python ./scalapack_MLA.py
```

See `https://github.com/gptune/GPTune/blob/master/examples/Scalapack-PDGEQRF/run_examples.sh` for a complete bash script. Note that these MPI runtime parameters are necessary for OpenMPI 4.0.1, higher versions have not been tested extensively.

As all the function evaluation data are check-pointed using JSON files (./gptune_db/PDGEQRF.json), GPTune is fault resilient. In case the tuning is interrupted due to timeout or code crashes, just rerun the above command and GPTune will continue tuning as expected. Alternatively, the RCI mode has even better fault resilience features as it will automatically skip function evaluations that crashed and continue to the next sample, see Section 4.3 for an example.

The following example runlog illustrates how to understand the code-generated information. First, the code prints out the $\mathbb{IS}$, $\mathbb{PS}$ and $\mathbb{OS}$. 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.

```
machine: Cori processor: haswell num_nodes: 1 num_cores: 32
IS: Space([Integer(low=128, high=1000, prior='uniform', transform='normalize'),
    Integer(low=128, high=1000, prior='uniform', transform='normalize')])
PS: Space([Integer(low=1, high=16, prior='uniform', transform='normalize'),
    Integer(low=1, high=16, prior='uniform', transform='normalize'),
    Integer(low=0, high=5, prior='uniform', transform='normalize'),
```

```
    Integer(low=1, high=32, prior='uniform', transform='normalize')])
OS: Space([Real(low=-inf, high=inf, prior='uniform', transform='identity')])
constraints : {'cst1': 'mb * p <= m', 'cst2': 'nb * nproc <= n * p', 'cst3': '
    nproc >= p'}
[HistoryDB] Create a JSON file at ./gptune.db/PDGEQRF.json

------Starting MLA with HistoryDB with 2 tasks and 40 samples each
MLA initial sampling NS1: 20
...
MLA iteration:  0
...
MLA iteration:  9
...

stats:  {'time_total': 51.9, 'time_fun': 24.7, 'time_search': 8.2, 'time_model':
    18.8, ... }

tid: 0
    m:1000 n:1000
    Ps  [[13, 7, 3, 3], [10, 1, 4, 5], [15, 2, 4, 4], [4, 10, 3, 7], [4, 12, 3,
    4], [3, 15, 5, 29], [16, 15, 2, 1], [15, 7, 0, 1], [14, 4, 1, 2], [15, 3, 5,
    3]]
    Os  [[0.0257], [0.0301], [0.022567], [0.033172], [0.030198], [0.046653],
    [0.054876], [0.105214], [0.043566], [0.019439]]
    Popt  [15, 3, 5, 3] Oopt  0.019439 nth  9
tid: 1
    m:500 n:500
    Ps  [[5, 9, 3, 3], [3, 7, 5, 4], [16, 3, 5, 3], [3, 1, 5, 4], [3, 10, 0, 1],
    [2, 5, 5, 30], [15, 6, 1, 1], [13, 2, 4, 3], [16, 4, 2, 2], [3, 14, 4, 16]]
    Os  [[0.009841], [0.009238], [0.007714], [0.008456], [0.030325], [0.017701],
    [0.246505], [0.009073], [0.00774], [0.014576]]
    Popt  [16, 3, 5, 3] Oopt  0.007714 nth  2
```

Listing 5: runlog of MLA

4.1.2 TLA (Type I)

Inserting the following code segments after line 186 of Listing 3, GPTune will call TLA in Section 6.2.5 that will run autotuning for a new task [[450,450]] while leveraging pre-collected function evaluations for tasks [[400,500],[800,600]].

As shown by the "LoadFunctionEvaluations" function, the user can read historical function evaluation data (an array of Python dictionaries) for each of the source tasks. The user can specify the TLA method using the GPTune options, e.g., "options["TLA_method"]="LCM". Since the TLA is an autotuning run, the user needs to (re-)initialize the GPTune instance with the new data object. Then the user can run the TLA_I call as shown by line 23.

```
# read source function evaluation data from the local database file
def LoadFunctionEvaluations(Tsrc):
    function_evaluations = [[] for i in range(len(Tsrc))]
    with open ("gptune.db/PDGEQRF.json", "r") as f_in:
        for func_eval in json.load(f_in)["func_eval"]:
            task_parameter = [func_eval["task_parameter"]["m"], func_eval["
    task_parameter"]["n"]]
            if task_parameter in Tsrc:
```

```
                function_evaluations[Tsrc.index(task_parameter)].append(func_eval)
    return function_evaluations

# options for the transfer learning
options["TLA_method"] = "LCM"
options["model_class"] = "Model_GPy_LCM"
options.validate(computer=computer)

# re-initialize the data object, the history database, and the GPTune instance for
    running transfer learning as a new autotuning run
data = Data(problem)
historydb=HistoryDB(meta_dict=tuning_metadata)
gt = GPTune(problem, computer=computer, data=data, options=options,historydb=
    historydb, driverabspath=os.path.abspath(__file__))

# Specify the new task parameter and run TLA_I
newtask = [[400, 5000]]
(data, modeler, stats) = gt.TLA_I(NS=nrun, Tnew=newtask,
    source_function_evaluations=LoadFunctionEvaluations(giventask))

""" Print all input and parameter samples """
for tid in range(len(data.I)):
    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].tolist())
    print('    Popt ', data.P[tid][np.argmin(data.O[tid])], 'Oopt ', min(data.O[
    tid])[0], 'nth ', np.argmin(data.O[tid]))
```

Listing 6: scalapack QR example: MLA+TLA_I

The following runlog illustrates what the output of TLA looks like. Please refer to Section 4.1.1 for the output of MLA to collect the source data for running this TLA. Note that one can also directly run TLA without running a fresh MLA experiment, as long as existing MLA tuning results have been stored in the database file.

```
------Starting TLA_I (LCM GPY) for 1 tasks and 10 samples each with 2 source tasks
...
tid: 0
    m:400 n:500
    Ps  [[8, 2, 2, 1], [11, 4, 3, 3], [3, 2, 1, 2], [8, 10, 3, 4], [4, 8, 4, 12],
    [9, 4, 0, 1], [10, 3, 4, 1], [15, 5, 4, 3], [13, 13, 2, 2], [15, 7, 1, 1]]
    Os  [[0.004799], [0.007173], [0.031857], [0.009178], [0.011421], [0.011601],
    [0.004715], [0.007649], [0.007863], [0.160016]]
    Popt  [10, 3, 4, 1] Oopt  0.004715 nth  6
```

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

4.1.3 TLA (Type II)

Inserting the following code segments after line 142 of Listing 3, GPTune will call TLA in Section 6.2.6 to predict the optimal tuning parameters for 1 new task [[400,500]]. The interface is similar to the TLA_I interface in Section 4.1.2. The differences are that TLA_II does not require to provide the number of samples for the new task (because TLA_II does not collect further function evaluations) and TLA_II requires the task parameter values for the source tasks (shown by "Tsrc" in Listing 8.

```
""" Call TLA_II for 1 new task using the constructed LCM model"""

# the data object initialized to run transfer learning as a new autotuning run
data = Data(problem)
historydb=HistoryDB(meta_dict=tuning_metadata)
gt = GPTune(problem, computer=computer, data=data, options=options,historydb=
    historydb, driverabspath=os.path.abspath(__file__))

# load source function evaluation data
def LoadFunctionEvaluations(Tsrc):
    function_evaluations = [[] for i in range(len(Tsrc))]
    with open ("gptune.db/PDGEQRF.json", "r") as f_in:
        for func_eval in json.load(f_in)["func_eval"]:
            task_parameter = [func_eval["task_parameter"]["m"], func_eval["
    task_parameter"]["n"]]
            if task_parameter in Tsrc:
                function_evaluations[Tsrc.index(task_parameter)].append(func_eval)
    return function_evaluations

newtask = [[400, 500]]
(aprxopts, objval, stats) = gt.TLA_II(Tnew=newtask, Tsrc=giventask,
    source_function_evaluations=LoadFunctionEvaluations(giventask))
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])
```

Listing 8: scalapack QR example: MLA+TLA.

The following runlog illustrates the output of this TLA run. Like the TLA_I example in Section 4.1.2, one can directly run TLA_II without running a fresh MLA run, if the source dataset (historical function evaluations) already exists in a database file.

```
------Starting TLA_II for task:  [[400, 500]]
...
[('QR', 400, 500, 1, 32, 128, 24, 1, 30, 3, 10, 1.0, 32)]  scalapack time:
    [0.0066]
stats:  {'time_total': 1.855111694, 'time_fun': 1.578588975}
new task: [400, 500]
    predicted Popt:  [16, 3, 5, 3]  objval:  [[0.0066]]
```

Listing 9: runlog of MLA+TLA (the part of MLA is skipped)

4.1.4 Sensitivity Analysis

Listing 10 below illustrates how to run a sensitivitiy analysis based on the function evaluation results obtained from the MLA run in Section 4.1.1. The user first needs to define the tuning problem space, $\mathbb{IS}$, $\mathbb{PS}$, and $\mathbb{OS}$, in a Python dictionary which corresponds to the problem space definition. Then, GPTune will filter the source function evaluations based on the problem space definition (e.g., disregard parameter configuration samples whose parameter value is out of the given range), and run the sensitivity analysis based on the collected function evaluations.

```
# Define the problem space in a Python dictionary
```

```
problem_space = {
    "input_space": [
        {"name":"m", "type":"integer", "transformer":"normalize",
         "lower_bound":mmin, "upper_bound": mmax},
        {"name":"n", "type":"integer", "transformer":"normalize",
         "lower_bound":nmin, "upper_bound": nmax}
    ],
    "parameter_space": [
        {"name":"mb", "type":"integer", "transformer": "normalize",
         "lower_bound":1, "upper_bound": 16},
        {"name":"nb", "type":"integer", "transformer": "normalize",
         "lower_bound":1, "upper_bound": 16},
        {"name":"lg2npernode", "type":"integer", "transformer": "normalize",
         "lower_bound":0, "upper_bound":5},
        {"name":"p", "type":"integer", "transformer": "normalize",
         "lower_bound":1, "upper_bound": 32}
    ],
    "output_space": [
        {"name":"r", "type":"real", "transformer":"identity",
         "lower_bound":float("-Inf"), "upper_bound":float("inf")}
    ]
}

# Run sensitivity analysis using function evaluations from the given DB file path
import gptune
ret = gptune.SensitivityAnalysis(
    problem_space = problem_space,
    modeler="Model_GPy_LCM",
    method="Sobol",
    input_task = [1000,1000],
    historydb_path = "gptune.db/PDGEQRF.json",
    num_samples=512)
print (ret)
```

Listing 10: ScaLAPACK QR example: running sensitivity analysis

We provide an example code for the sensitivity analysis of the PDGEQRF example, in `https://github.com/gptune/GPTune/blob/master/examples/Scalapack-PDGEQRF/scalapack_sensitivity_analysis.py`. To run the sensitivity analysis, the user first needs to prepare performance data (function evaluations) in the database file (e.g., gptune.db/PDGEQRF.json). Then, the user can run the Python example code, i.e., $ python scalapack_sensitivity_analysis.py.

The following runlog illustrates what the output of the sensitivity analysis can look like.

```
Run Sensitivity Analysis
{'S1': {'mb': 0.05079833404269942, 'nb': 0.00784196972311793, 'lg2npernode':
    0.8149509775738293, 'p': 0.0075335088763804825}, 'S1_conf': {'mb':
    0.03853778678769951, 'nb': 0.0139738595481947, 'lg2npernode':
    0.12359959066368517, 'p': 0.01742636065611316}, 'S2': {'mb': {'mb': nan, 'nb':
    -0.006777139696287966, 'lg2npernode': 0.09337717894944186, 'p':
    -0.00791248729390076}, 'nb': {'mb': nan, 'nb': nan, 'lg2npernode':
    0.01414975749821945, 'p': 0.005043118017565775}, 'lg2npernode': {'mb': nan, 'nb
    ': nan, 'lg2npernode': nan, 'p': 0.012061686530822187}, 'p': {'mb': nan, 'nb':
    nan, 'lg2npernode': nan, 'p': nan}}, 'S2_conf': {'mb': {'mb': nan, 'nb':
    0.06662559350375566, 'npernode': 0.10982750577252501, 'p':
    0.06796876808809083}, 'nb': {'mb': nan, 'nb': nan, 'lg2npernode':
    0.03266212446811037, 'p': 0.020238036469397303}, 'lg2npernode': {'mb': nan, 'nb
```

```
': nan, 'lg2npernode': nan, 'p': 0.17073352326295915}, 'p': {'mb': nan, 'nb':
nan, 'lg2npernode': nan, 'p': nan}}, 'ST': {'mb': 0.14414079154179915, 'nb':
0.01548033174306157, 'lg2npernode': 0.9342346332100098, 'p':
0.01831622825026556}, 'ST_conf': {'mb': 0.032538721858522114, 'nb':
0.0027290251041252383, 'lg2npernode': 0.1219051523063763, 'p':
0.0043049424167933064}}
```

Note that, the validity of the sensitivity analysis result depends on the number of historical function evaluation samples and the goodness-of-fit of the surrogate model (fitting quality). The example runlog used 100 historical function evaluations obtained a previous MLA call (50 initial pilot samples and 50 sequential samples).

4.2 SuperLU_DIST

For a typical SuperLU_DIST driver, one can use a given sparse matrix to define a task, and consider (COLPERM, LOOKAHEAD, npernode, 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, npernode is the MPI count per compute node, 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 an 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.

4.2.1 Preparing the meta JSON file

Just like *PDGEQRF*, a meta JSON file located at ./gptune/meta.json needs to be edited for SuperLU_DIST. One can also use `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST/run_examples.sh` to generate the meta JSON file.

4.2.2 MLA and TLA (Type II)

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 the complete working codes at

- The GPTune Python driver: `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST/superlu_MLA.py`
- The C driver from the application side. `https://github.com/xiaoyeli/superlu_dist/blob/master/EXAMPLE/pddrive_spawn.c`
- The bash script to run the tests: `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST/run_examples.sh`

```
def objectives(point):
  nodes = point['nodes']
  cores = point['cores']
  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']
  npernode = 2**point['npernode']
  nproc = nodes*npernode
  nthreads = int(cores / npernode)

  NSUP = point['NSUP']
  NREL = point['NREL']
  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)
  info.Set('npernode','%d'%(npernode))  # YL: npernode is deprecated in openmpi 4.0, but no other parameter (e.g. 'map-by') works

  """ 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():
  ntask = 2   # 2 tasks used for MLA
  NS = 20  # 20 samples per task
  matrices = ["big.rua", "g4.rua", "g20.rua"]
  # read the meta.json file
  (machine, processor, nodes, cores) = GetMachineConfiguration()
  print ("machine: " + machine + " processor: " + processor + " num_nodes: " + str(nodes) + " num_cores: " + str(cores))

```

```
""" 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")
npernode  = Integer     (0, int(log2(cores)), transform="normalize", name="
  npernode")
NSUP      = Integer     (30, 300, transform="normalize", name="NSUP")
NREL      = Integer     (10, 40, transform="normalize", name="NREL")
PS = Space([COLPERM, LOOKAHEAD, npernode, nprows, NSUP, NREL])
# Output
time   = Real        (float("-Inf") , float("Inf"), transform="normalize", name=
  "time")
OS = Space([time])
# Constraints
cst1 = "NSUP >= NREL"
cst2 = "nodes * 2**npernode >= nprows"
constraints = {"cst1" : cst1, "cst2" : cst2}
constants={"nodes":nodes,"cores":cores}
print(IS, PS, OS, constraints)
problem = TuningProblem(IS, PS, OS, objectives, constraints, None, constants)
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
gt = GPTune(problem, computer = computer, data = data, options = options,
  driverabspath=os.path.abspath(__file__))

""" Build MLA with the given list of tasks """
giventask = [["g4.rua"], ["g20.rua"]]
NI = len(giventask)
(data, model,stats) = gt.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].tolist())
    print('    Popt ', data.P[tid][np.argmin(data.O[tid])], 'Oopt ', min(data.O[
```

```
    tid])[0], 'nth ', np.argmin(data.O[tid]))

  """ Call TLA for a new task using the constructed LCM model"""
  newtask = [["big.rua"]]
  (aprxopts,objval,stats) = gptune.TLA_II(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 11: superlu_MLA.py.

```
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. */
  //...

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

Listing 12: pddrive_spawn.c.

4.2.3 Multi-objective MLA

The following example demonstrates the capability of multi-objective autotuning feature of GPTune with two objectives. As discussed in Section 3.1.2, GPTune's (default) multi-objective tuning focuses on optimizing multiple objectives by achieving the Pareto front with support for specifying output constraints on each of the objectives. In this example, we focus on optimizing both runtime and memory consumption of a sparse LU factorization. The example calls MLA to build an 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 the complete working codes at

- The GPTune Python driver: `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST/superlu_MLA_MO.py`
- The C driver from the application side. `https://github.com/xiaoyeli/superlu_dist/blob/master/EXAMPLE/pddrive_spawn.c`
- The bash script to run the tests: `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST/run_examples.sh`

```
def objectives(point):
  nodes = point['nodes']
  cores = point['cores']
  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']
    npernode = 2**point['npernode']
    nproc = nodes*npernode
    nthreads = int(cores / npernode)
    NSUP = point['NSUP']
    NREL = point['NREL']
    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)
    info.Set('npernode','%d'%(npernode))  # YL: npernode is deprecated in openmpi 4.0, but no other parameter (e.g. 'map-by') works

    """ 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[0], tmpdata[1]]



def main():
    ntask = 3   # 3 tasks used for MLA
    NS = 20  # 20 samples per task
    matrices = ["big.rua", "g4.rua", "g20.rua"]
    (machine, processor, nodes, cores) = GetMachineConfiguration()
    print ("machine: " + machine + " processor: " + processor + " num_nodes: " + str(nodes) + " num_cores: " + str(cores))


    """ 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")
npernode  = Integer     (0, int(log2(cores)), transform="normalize", name="
  npernode")
NSUP      = Integer     (30, 300, transform="normalize", name="NSUP")
NREL      = Integer     (10, 40, transform="normalize", name="NREL")
PS = Space([COLPERM, LOOKAHEAD, npernode, 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 = "nodes * 2**npernode >= nprows"
constraints = {"cst1" : cst1, "cst2" : cst2}
constants={"nodes":nodes,"cores":cores}
print(IS, PS, OS, constraints)
problem = TuningProblem(IS, PS, OS, objectives, constraints, None, constants)
computer = Computer(nodes = nodes, cores = cores, hosts = None)

""" 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
gt = 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) = gt.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 13: superlu_MLA_MO.py.

In the above code example, one can add constraints on the objectives, for example by:

```
# Two-objective tuning, with the constraint that the runtime should be less than
    50 seconds
runtime  = Real (float("-Inf") , 50.0, transform="normalize", name="r")
memory  = Real (float("-Inf") , float("Inf"), transform="normalize", name="memory"
    )
```

or

```
# Single-objective tuning (memory only), but the runtime needs to be below 50
    seconds
runtime  = Real (float("-Inf") , 50.0, transform="normalize", name="r", optimize=
    False)
memory  = Real (float("-Inf") , float("Inf"), transform="normalize", name="memory"
    )
```

4.3 SuperLU_DIST (RCI)

For the RCI mode (see Section 3.2), in addition to the GPTune Python driver, the user also needs to provide a GPTune bash driver. The bash driver will keep querying the GPTune Python driver for next sampling points, get the sampling points for the dababase file, invoke the application, and write the results back into the dabasefile. This process continues until no more sample required. Following the database format described in Section 3.3.4, "time: null, memory: null" will appear under "evaluation_result: ", indicating this is a sample requiring evaluation. As the application is invoked from bash, it does not need to be compiled using the same software dependence as GPTune, hence OpenMPI is not required. In addition, the modification in Section 6.1.1 is also not needed. For the SuperLU_DIST application, all tuning and task parameters can be passed to the application via environment variables and command line options, and the objective function evaluations can be obtained by searching the runlog.

4.3.1 Preparing the meta JSON file

Just like *PDGEQRF*, a meta JSON file located at ./gptune/meta.json needs to be edited for SuperLU_DIST. One can also use `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST_RCI/run_examples.sh` to generate the meta JSON file.

4.3.2 Multi-objective MLA in RCI mode

The following example performs the same tuning experiment as Section 4.2.3. In other words, the example demonstrates the capability of multi-objective autotuning feature of GPTune (in the RCI mode) 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 the bash scripts keeps calling superlu_MLA_MO_RCI.py which is similar to the script superlu_MLA_MO.py in Section 4.2, except that superlu_MLA_MO_RCI.py does not define the objective function (as it's directly executed in the bash script), and options['RCI_mode']=True is required. Note that only the simplified code is shown here, please refer to the complete working codes at

- The GPTune Python driver: `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST_RCI/superlu_MLA_MO_RCI.py`
- The GPTune Bash driver: `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST_RCI/superlu_MLA_MO_RCI.sh`
- The bash script to run the tests: `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST_RCI/run_examples.sh`

```
start=`date +%s`
nrun=20 # number of samples per task
# name of your machine, processor model, number of compute nodes, number of cores per compute node, which are defined in .gptune/meta.json
declare -a machine_info=($(python -c "from gptune import *;
(machine, processor, nodes, cores)=list(GetMachineConfiguration());
print(machine, processor, nodes, cores)"))
machine=${machine_info[0]}
processor=${machine_info[1]}
nodes=${machine_info[2]}
cores=${machine_info[3]}

obj1=time    # name of the first objective defined in the python file
obj2=memory    # name of the second objective defined in the python file

database="gptune.db/SuperLU_DIST.json"  # the phrase SuperLU_DIST should match the application name defined in .gptune/meta.jason

# start the main loop
more=1
while [ $more -eq 1 ]; do

    # call GPTune and ask for next sample points
    python ./superlu_MLA_MO_RCI.py -nrun $nrun

    # check whether GPTune needs more data
    idx=$( jq -r --arg v0 $obj1 '.func_eval | map(.evaluation_result[$v0] == null) | index(true) ' $database )
    if [ $idx = null ];then; more=0; fi;

    # if so, call the application code (GPTune can requires mutiple samples to be evaluated)
```

```
    while [ ! $idx = null ]; do
        echo " $idx"    # idx indexes the record that has null objective function values

        # use jq to retrieve task and tuning parameters
        declare -a input_para=($( jq -r --argjson v1 $idx '.func_eval[$v1].task_parameter' $database | jq -r '.[]'))
        declare -a tuning_para=($( jq -r --argjson v1 $idx '.func_eval[$v1].tuning_parameter' $database | jq -r '.[]'))

        # get the task input parameters, the parameters should follow the sequence of definition in superlu_MLA_MO_RCI.py
        matrix=${input_para[0]}

        # get the tuning parameters, the parameters should follow the sequence of definition in superlu_MLA_MO_RCI.py
        COLPERM=${tuning_para[0]}
        LOOKAHEAD=${tuning_para[1]}
        npernode=${tuning_para[2]}
        nprows=${tuning_para[3]}
        NSUP=${tuning_para[4]}
        NREL=${tuning_para[5]}

        # set environment variables and command line options
        npernode=$((2**$npernode))
        export OMP_NUM_THREADS=$(($cores / $npernode))
        export NREL=$NREL
        export NSUP=$NSUP
        nproc=$(($nodes*$npernode))
        npcols=$(($nproc / $nprows))
        RUNDIR="../SuperLU_DIST/superlu_dist/build/EXAMPLE"
        INPUTDIR="../SuperLU_DIST/superlu_dist/EXAMPLE/"

        # run the application, this doesn't have to be openmpi-compiled code
        mpirun -n $nproc $RUNDIR/pddrive_spawn -c $npcols -r $nprows -l $LOOKAHEAD -p $COLPERM $INPUTDIR/$matrix | tee a.out

        # get the result (for this example: search the runlog)
        result1=$(grep 'Factor time' a.out | grep -Eo '[+-]?[0-9]+([.][0-9]+)?')
        result2=$(grep 'Total MEM' a.out | grep -Eo '[+-]?[0-9]+([.][0-9]+)?')

        # use jq to write the data back to the database file
        jq --arg v0 $obj1 --argjson v1 $idx --argjson v2 $result1 '.func_eval[$v1].evaluation_result[$v0]=$v2' $database > tmp.json && mv tmp.json $database
        jq --arg v0 $obj2 --argjson v1 $idx --argjson v2 $result2 '.func_eval[$v1].evaluation_result[$v0]=$v2' $database > tmp.json && mv tmp.json $database

        # get the next sample to be evaluated
        idx=$( jq -r --arg v0 $obj1 '.func_eval | map(.evaluation_result[$v0] == null) | index(true) ' $database )
    done
done
end=`date +%s`
runtime=$((end-start))
```

```
74 echo "Total tuning time: $runtime"
```

Listing 14: superlu_MLA_MO_RCI.sh.

5 Tuning Results

5.1 Parallel speedups of GPTune

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

$$y_{demo}(t,x) = \exp\big(-(x+1)^{t+1}\big)\cos(2\pi x)\sum_{i=1}^{3}\sin\big(2\pi x(t+2)^i\big) \tag{17}$$

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. 6 plots $y_{demo}(t,x)$ versus x for four different values of t and marks the minimum objective function values.

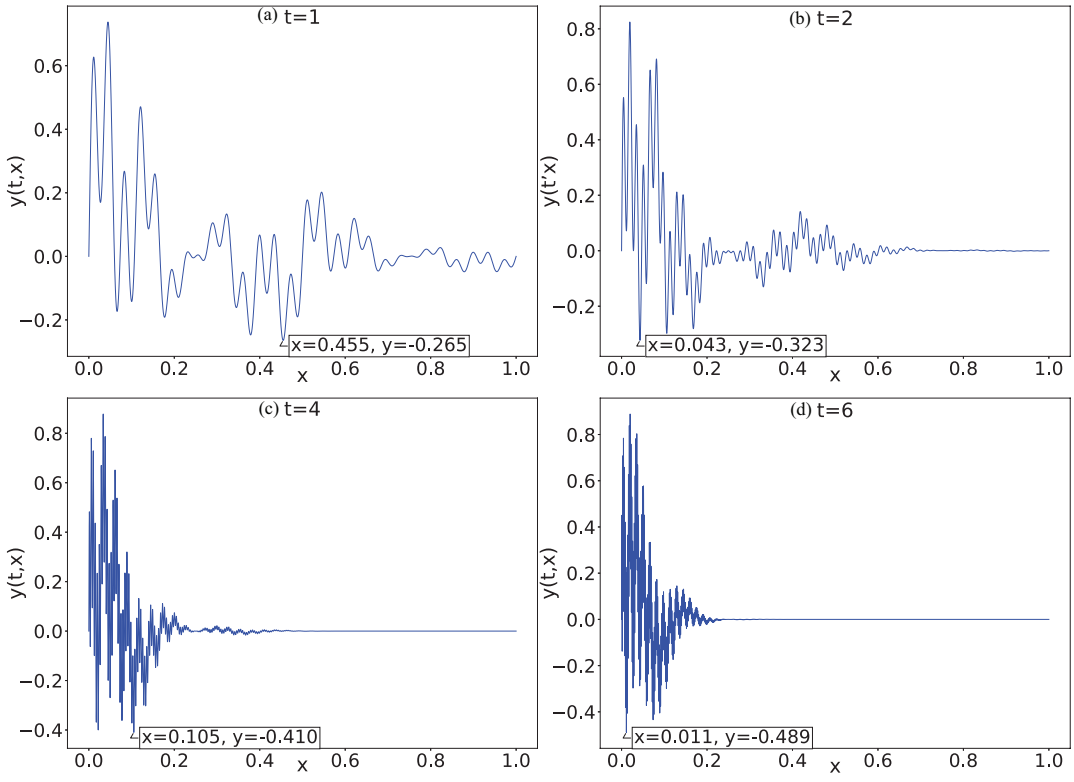

Figure 6: The objective functions in (17) for four task parameter values t.

First, we evaluate the parallel performance of the MLA algorithm on a Threadripper 1950X 16-core processor using $n_i = 20$ tasks. In Fig. 7, 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 search phases, respectively. More results can be found in [26].

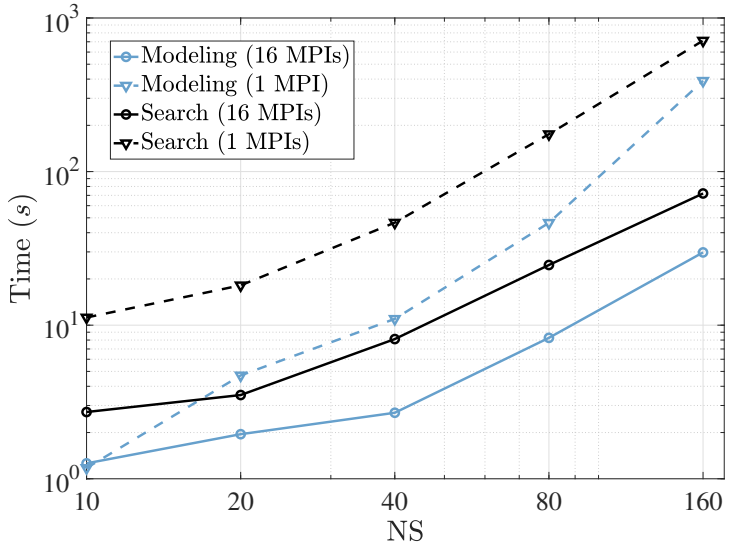

Figure 7: Modeling and search time for 1 and 16 MPIs using the objective function f_{demo}.

Example script. `https://github.com/gptune/GPTune/blob/master/examples/GPTune-Demo/demo_parallelperformance.py`

5.2 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), $\tilde{y}_2(t, x) = 10 y_{demo}(t, x)$ (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 4 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. 6(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 cheap performance models, GPTune requires significantly fewer samples to build an accurate LCM model. More results can be found in [26].

Example script. `https://github.com/gptune/GPTune/blob/master/examples/GPTune-Demo/demo_perf_model.py`

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 4: 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 5: Runtime of different phases in the GPTune single-objective and multi-objective MLA with a total of 400 samples.

5.3 Efficiency of multi-task learning

Next, we use the ScaLAPACK QR example in Section 4.1.1 to compare the performance of the GPTune MLA algorithms with single-task (n_i=1) and multi-task (n_i=20) settings. We use 16 NERSC Cori nodes assuming a fixed budget of $n_i \times \text{NS} = 400$ and NS1 = NS/2. For n_i=1, we consider the task ($m = 4674, n = 3608$); for n_i=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 5 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. 8 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 n_i=20 and the blue dots correspond to n_i=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. More results can be found in [26].

Example script. `https://github.com/gptune/GPTune/blob/master/examples/Scalapack-PDGEQRF/scalapack_MLA.py`

5.4 Capability of multi-objective tuning

Next, we illustrate the multi-objective feature of GPTune for tuning the factorization time and memory in SuperLU_DIST [24]. As explained in Section 4.2, 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 [23] using 8 NERSC Cori nodes.

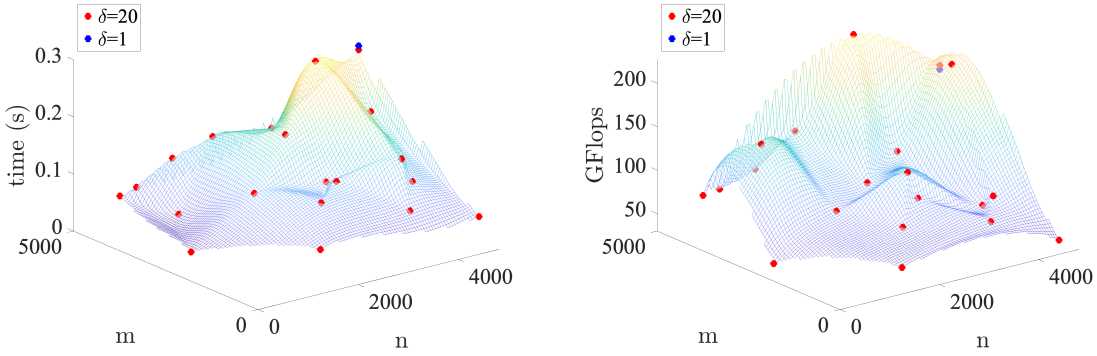

Figure 8: (a) Runtime (the objective) and (b) GFlops for 20 tasks of QR factorization after autotuning.

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 6 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. 9 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. More results can be found in [26].

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

Example script. `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST/superlu_MLA_MO.py`

5.5 Advantage of multi-fidelity, multi-task tuning

Here we demonstrate the feature of multi-fidelity and multi-task tuning from GPTuneBand using the Hypre application.

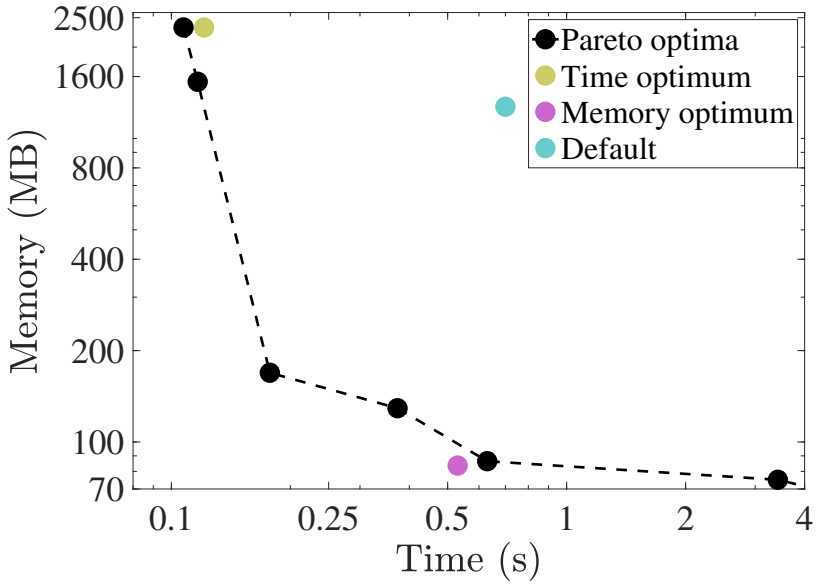

Figure 9: Logarithmic plots of the optimal objective functions values (factorization time and memory of SuperLU_DIST with 8 NERSC Cori nodes) found by GPTune using single-objective and multi-objective tuning. The objective function values using the default tuning parameters are also plotted.

Hypre [10] contains several families of parallel algebraic multigrid preconditioners and solvers for large-scale sparse linear systems. Here we tune the runtime of GMRES with the BoomerAMG preconditioner for solving the convection-diffusion equation on structured 3D grids. Task parameters are defined by the a, c coefficients in the convection-diffusion equation: $-c\Delta u + a\nabla \cdot u = f$. There are 12 tuning parameters of integer, real and categorical types, for example processor topology parameters, total number of MPIs, the AMG strength threshold, the type of the parallel coarsening algorithm and the type of parallel interpolation operator.

The fidelity level is defined by the discretization with k^3 grid points, where k ranges from $k_{\min} = 10$ to $k_{\max} = 100$. Given the $\mathcal{O}(k^3)$ computational complexity of the algebraic multigrid algorithm, the fidelity mapping from b (i.e., $B(s)$ in Algorithm 5) to k is a linear function interpolating $[b_{\min}, k_{\min}^3]$ and $[b_{\max}, k_{\max}^3]$. Experiments are run on 2 NERSC Cori nodes, with 5 repeated runs and GPTuneBand uses NLOOP=1 pass. The total equivalent number of samples are matched across the 5 tuners. Each highest-fidelity evaluation takes 5 seconds on average, depending on the value of the tuning parameter. Fig. 10 shows that GPTuneBand constantly overperforms the other tuners. More detailed explanations can be found in [41].

Example script. `https://github.com/gptune/GPTune/blob/master/examples/Hypre/hypre_MB.py`

5.6 Effectiveness of transfer learning

Here, we provide a demonstration of the effectiveness of transfer learning using the PDGEQRF tuning example.

Figure 11 compares the tuning result from SLA (single-task learning) and TLA for 20 function

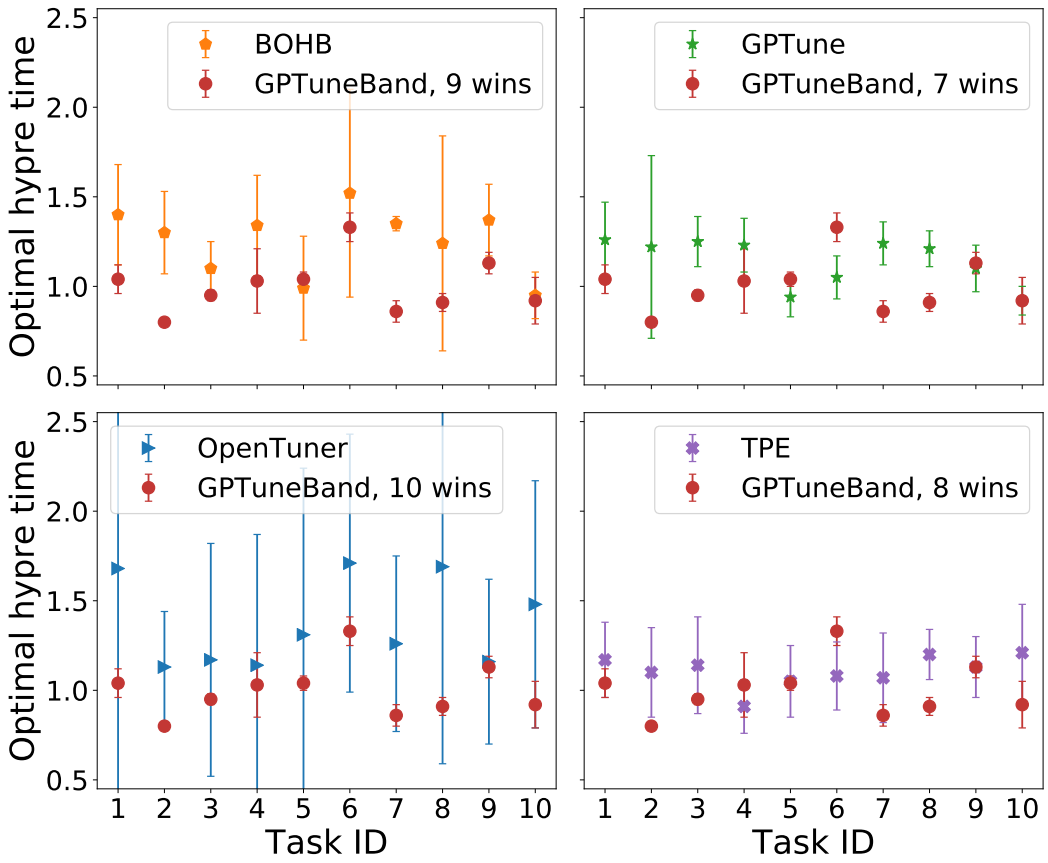

Figure 10: Best Hypre runtime of 10 tasks averaged over 5 repeated tuning experiments with 5 different tuners: GPTune (multi-task, single-fidelity), GPTuneBand (multi-task, multi-fidelity), HpBandSter-TPE (single-task, single-fidelity), HpBandSter-BOHB (single-task, multi-fidelity), and OpenTuner (single-task, single-fidelity). One pass (NLOOP=1) is used in GPTuneBand

evaluations. The experiments used 8 nodes (256 cores) and 64 nodes (2,048 cores) on Cori Haswell, and we report the average from three tuning experiments. As shown in the figure, TLA improves the tuning results with a significant margin, especially on the 64 Cori node setting. The experiments consider three tasks [10000,10000], [20000,20000], [30000,30000]. In particular, for this experiment, we used the LCM-based TLA with a pre-trained (black-box) surrogate model for the source task (with 19 function evaluations); therefore we used the TLA option, `TLA_method="LCM_BF"` (see Section 3.1.4 for the details of this option and for the information of other available TLA options). We considered one source task, one of these considered tasks, except the given task. TLA (best) and TLA (worst) represent the best and worst case from choosing different tasks, which implies it can be important to choose an appropriate dataset for transfer learning. For the details of this experiment, please refer to [5].

Example script. `https://github.com/gptune/GPTune/blob/master/examples/ScaLAPACK-PDGEQRF-ATMG/scalapack_TLA_task.py`

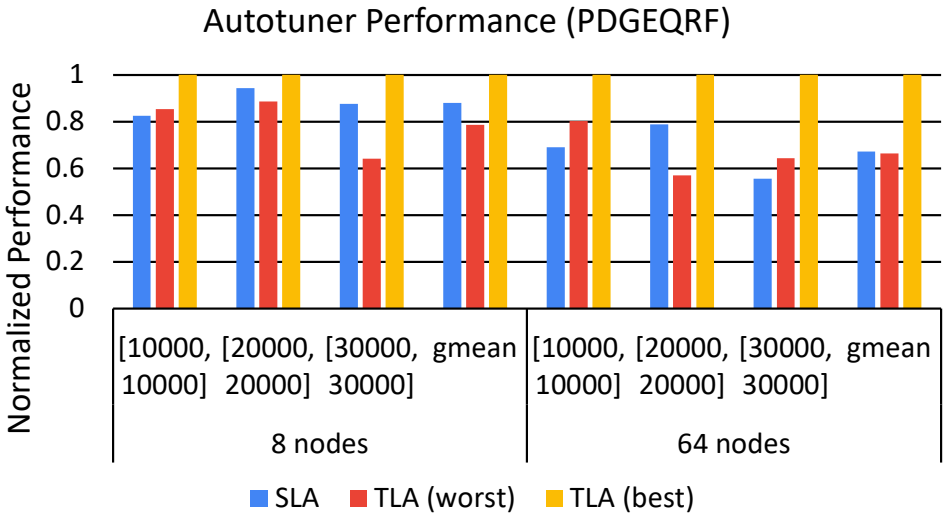

Figure 11: Demonstration of the effectiveness of TLA for PDGEQRF tuning.

5.7 Detection of non-smoothness

Here we demonstrate the capability of cGP for tuning non-smooth objective functions. For details of this feature, please refer to [28]. We consider a simple 1D function $y = f(x)$ for matrix-matrix multiplication, where x is the block size, and y is the measured flops.

As shown in the Figure 12,the cGP method can be applied to differentiate non-smoothness regimes. In the (large) blocked matrix multiplication for a 1000 by 1000 matrix (matmul), the performance (in terms of flops) would have drops when the block size exceeds the memory cache. We sequentially sample 10 and 90 samples (correspoding to 10 and 90 different block sizes when performing matrix multiplications) and then fit the GP surrogate and cGP surrogate, respectively. The cGP has a better detailed fit even with smaller sample sizes and illustrates different drops, which correspond to the physical memory cache size calculation closely.

In short, the cGP can (i) detect potential change of regimes with very limited samples, and (ii) fit the details of the black-box function better compared to regular GP, capturing non-stationary features. We suggest its usage when the usual smooth kernels are not working as expected or non-smoothness exists in the black-box function.

Example script. `https://github.com/gptune/GPTune/blob/master/examples/SuperLU_DIST/superlu_MLA.py`

5.8 Handling categorical and mixed variables

For details of this feature, please refer to [27]. For HPC applications like STRUMPACK [15], a single execution is very expensive and hence needs to have optimal configuration of 3 categorical and 2 continuous (hyper-)parameters. In Figure 13 we compared the hybrid model against other state-of-the-art methods for mixed variables (MAB methods like [34] are not applicable because the per category reward is too large; MCTS with independent GPs like [32] are not applicable because the GP per category would have almost no samples). Predominantly, our method outperforms

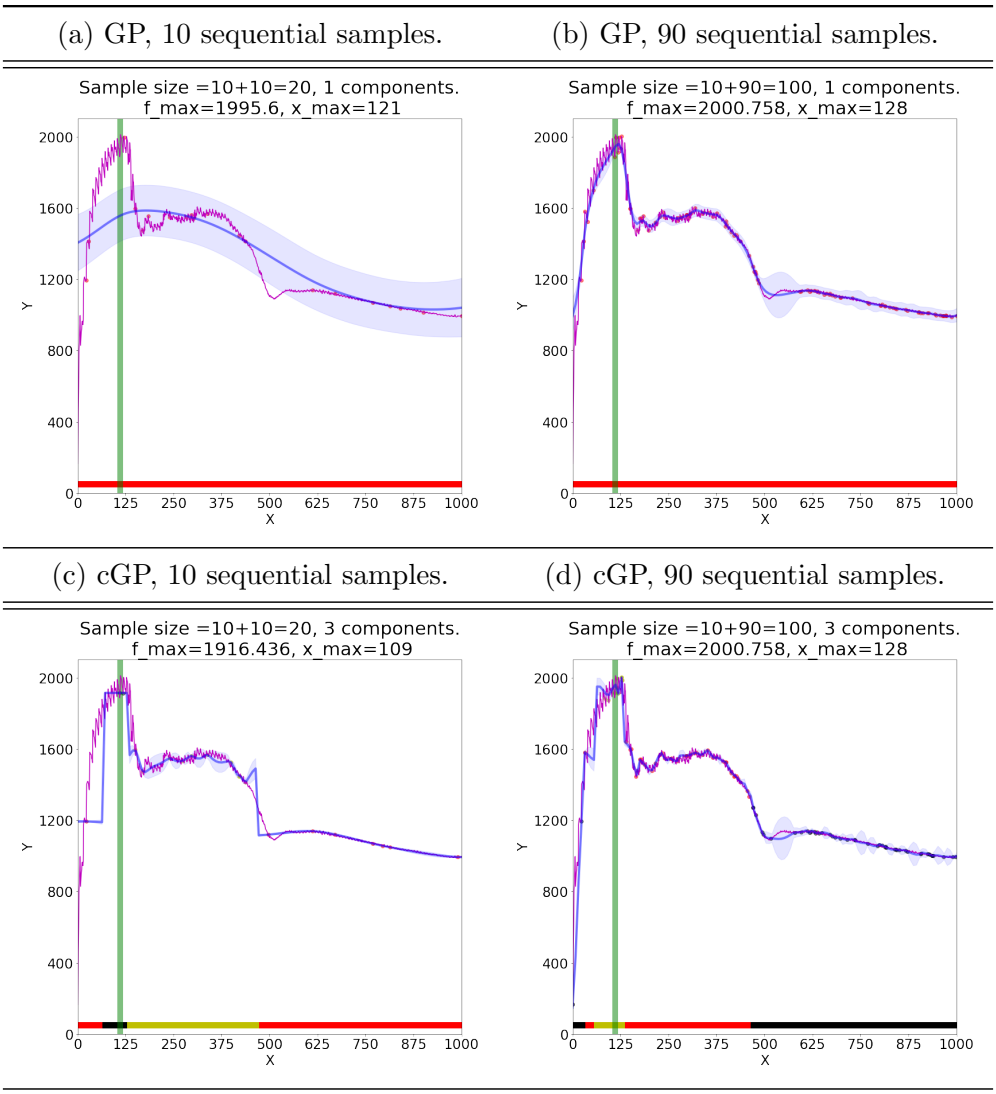

Figure 12: The computational speed function and fitted surrogate models. The dark blue solid line and light blue shaded area are the mean and variance of fitted prediction models. $(\boldsymbol{x}_{\max}, f_{\max})$ records the optimal block size and the actual optimal speed as illustrated by the green vertical line on the figure. Bottom colored bars with different colors indicate to which cluster certain portions of the domain belong (but color has no semantic meaning). We also use magenta solid lines to overlay the truth for comparison purposes. (The exploration rates of these experiments are all 1.)

all the other competitors in a much smaller number of iterations. This method is intrinsically memory-efficient due to the MCTS definition, and its capability of choosing kernels dynamically also allows us to pin down the model earlier, empirically.

In short, the hybrid model (i) can handle mixed variable black-box functions more efficiently, and (ii) dynamically selects different continous-categorical kernels in a GP surrogate, capturing continuous-categorical dependence better. We suggest its usage when there are different types of

variables in the input of black-box function as described and differentiated in Table 3. Currently, the hybrid model can be used for arbitrary dimensional input and univariate response output, but its multi-task generalization remains future work.

Example script. `https://github.com/gptune/GPTune/blob/master/examples/STRUMPACK/strumpack_MLA_Poisson3d_simple.py`

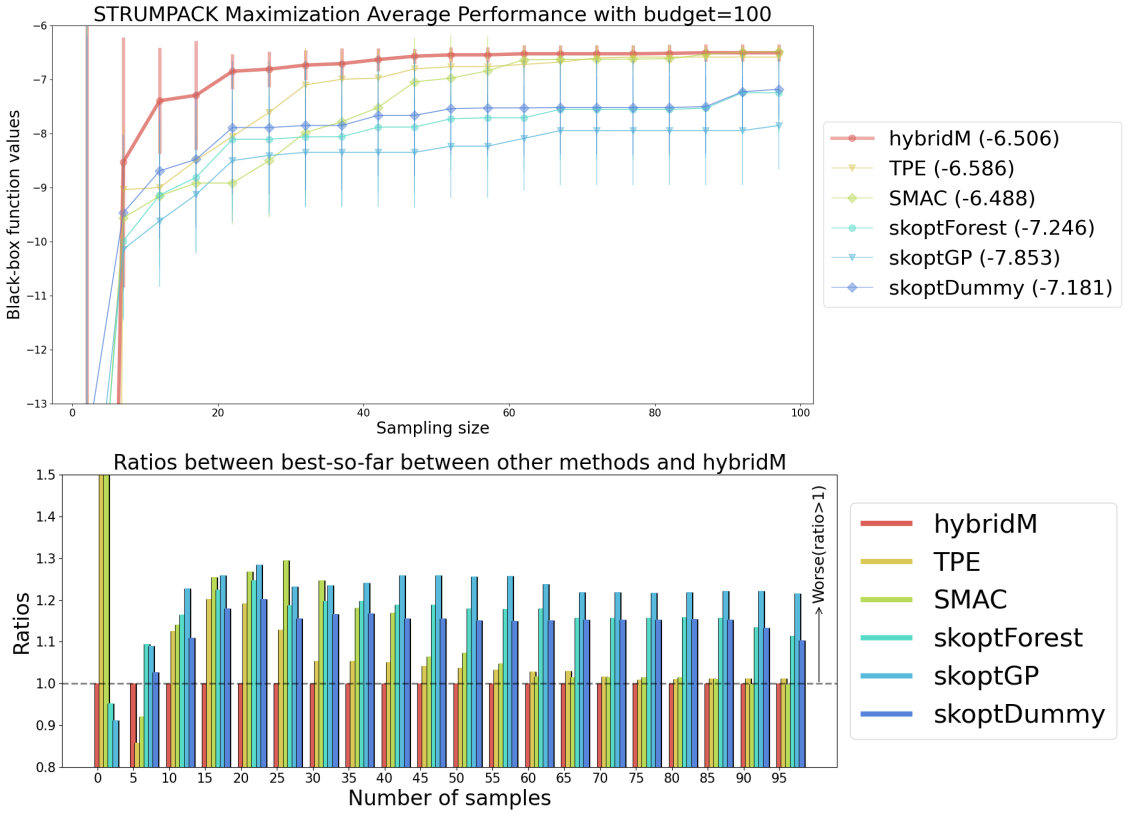

Figure 13: Comparison of performance between different methods (including skoptForest, skoptGP, skopyDummy [18], TPE [1] and SMAC [20]) on the HPC application of STRUMPACK [15] for Poisson3d with grid size 100. In the line plot, the mean of the optima is computed for each method; the standard deviation is indicated by the vertical segments. Note that we use negative optimal execution time to visualize (higher is better).

6 User Interface

This section describes the user interfaces for a GPTune driver along with utility functions to help analyze the obtained function evaluations. 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 [3].

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, MPI count per node, number of row processes, denoted by $(mb, nb, npernode, p)$ be the parameters to be tuned assuming a fixed number

of compute nodes (*nodes*) and cores per node (*cores*). Recall that other arguments affecting the runtime such as the number of OpenMP threads per MPI process *nthreads* (used in BLAS; for one MPI process, by default one thread is mapped to one core) and number of column processes q, can be calculated as $nthreads = \lfloor cores/npernode \rfloor$ and $q = \lfloor nodes * npernode/p \rfloor$. Note that we use $lg2npernode$ instead of *npernode* such that $npernode = 2^{lg2npernode}$ to enforce that *npernode* is power of 2. Finally, let (r) be the QR runtime with the given task parameters and tuning parameters.

6.1 Tuning setup

6.1.1 Modify the application code (only needed for MPI spawning mode)

If the user uses GPTune's MPI spawning mode (Fig. 2 (a) in Section 3.2) and the application is distributed-memory parallel, the user may need to make slight modifications to the application code to communicate with the GPTune tuning driver. Otherwise, no modification is required and the user may skip this subsection.

To modifiy a distributed-memory parallel code for the MPI spawning mode, for example, assuming the application code is written in the C language, the user needs to insert `MPI_Comm_get_parent` and `MPI_Comm_disconnect` after `MPI_Init` and before `MPI_Finalize`. The following example assumes that the parameters are passed in via the command line options (i.e., argv) and the function values are written into a file. For other options of passing parameters and returning objectives, see Section 3.2.2.

```
int main(int argc, char *argv[]){
MPI_Init( &argc, &argv );
MPI_Comm parent; MPI_Comm_get_parent(&parent); /* Get the inter-communicator. */
.../* Read parameters from argv, compute the objective and write it to a file. */
MPI_Comm_disconnect(&parent); /* Disconnect the inter-communicator. */
MPI_Finalize();}
```

6.1.2 Define the objective function to be minimized

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

Input:

- **point** [Dict]: 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 6.1.6 for their definitions. The dictionary can also contain point["C"] where "C" is a global constant (e.g., total node or core counts) passed to GPTune, see Section 6.1.7 for its definition.

Output:

- **res** [list of real values of length γ]: Array containing the objective function value, γ denotes dimension of the output space $\mathbb{OS}$.

- **additional_output** [Dict]: A Python dictionary that can contain any additional information to be recorded in the database for each function evaluation. Users can optionally use this return parameter to store useful information such as execution times of the multiple runs in a single function evaluation and detailed performance profiling results.

QR EXAMPLE:
point["m"], point["n"] are the task parameter arguments, point["mb"], point["nb"], point["npernode"], point["p"] are the tuning parameter arguments, point["nodes"], point["cores"], point["bunit"] are the global constants. The user is responsible for writing a driver code that writes the parameters to an input file, calls $PDGEQRF$ and reads the runtime from an output file and returns the runtime in **res** with $\gamma = 1$, e.g., using a MPI spawning approach. See Section 5 for more details.

6.1.3 Define the performance models

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

Input:

- **point** [Dict]: 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 6.1.6 for their definitions. Note that the input "point" is exactly the same as that for "objectives".

Output:

- **res** [list of real numbers of length $\tilde{\gamma}$]: 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"] are the task parameter arguments, point["mb"], point["nb"], point["npernode"], point["p"] are the tuning parameter arguments, point["nodes"], point["cores"], point["bunit"] are the global constants. The user can use a simple performance model **res**= $[mn^2/npernode/nodes/PF]$, with PF denoting the peak flop rate of the machine. Note that here mn^2 roughly denotes the total number of floating operations without a constant prefactor as we only need the performance model output to be on the same order as the objective function.

6.1.4 Define the performance models update

```
# define the cheap performance models update function
def models_update(data):
for i in range(len(data.I)):
# update the hyperparameters data.D[i] of the performance model for each task uing data.I[i], data.O[i], data.P[i]
```

Input:

- **data** [Class] (input/output): The data class contains all tuning parameter configurations, function evaluations, and possible hyperparameters for each task. See Section 6.1.11 for more details.

Output:

- **data** [Class]: The data class after updating the hyperparameters of the user-given performance model.

QR EXAMPLE:
For the performance model **res**= $[mn^2/npernode/nodes/PF]$, one can either use a constant PF, or a dynamic hyperparameter data.D[i]["PF"] per task i, which can be udpated using the growing sized samples.

6.1.5 Initialize the history database

```
historydb = HistoryDB(meta_dict, meta_path)
# initialize the history database class from ``./.gptune/meta.json" (default), a user-provided meta file meta\_path, or a user-provided dictionary meta\_dict defining the metadata.
```

Input:

- **meta_dict** [Dict]: The dictionary used to define the metadata (Section 3.3.2 explains the syntax of metadata description). Default to None.
- **meta_path** [string]: Absolute path of the file containing meta file. Default to None.

Output:

- **historydb** [Class]: The history database class recording machine information, software dependency, performance data and surrogate models.

6.1.6 Define the tuning spaces

This section discusses how users can define the tuning space, i.e., defining task parameter, tuning parameter, output space, global constant variables, and tuning constraints.

```
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
constants = {"const1" : const1, ...} # define global constant variables
```

- ***I_j*, *P_j*** [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. 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 $[0, 1)$. Specifically, the transform first converts the categorical data to its one-hot encoding, which is a $1 \times \alpha$ binary array with only element equal to 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. Note that, the "categories" in this interface should be a list of strings.

Please refer to `https://scikit-optimize.github.io/stable/modules/classes.html?highlight=space#module-skopt.space.space` for the scikit-optimize spaces, and their modified versions used by GPTune `https://github.com/gptune/GPTune/blob/master/patches/scikit-optimize/space.py`. Please refer to `https://scikit-optimize.github.io/stable/modules/generated/skopt.space.transformers.Normalize.html` for more details about the transform.

- ***O_j*** [Scikit-optimize spaces]: Supported spaces for O_j:
  - O_j = `Real(low, high, name="`O_j`", optimize=True)`. Note: if low and high are unknown, they can be set to float("-Inf") and float("Inf"), respectively. Here, "optimize" denotes whether this variable is to be optimized or not. If set to False, it means this objective needs to be bounded by [low, high], but does not need to be optimized. The "optimize" parameter can be used for constrained multi-objective tuning. For example, to optimize runtime while satisfying a certain accuracy level, one can define two objectives for the runtime and the error, and set "optimize=True" for the runtime objective and "optimize=False" for the error objective with a specific error range [low, high].
- **const1** [integer, real, or string]: define a global constant variable.
- **cst1** [string or function]: define one constraint using (a) a string, e.g., as cst1 = "$I_1 + P_2 < C$", or (b) a callable function, e.g., as

```
# define a constraint function
def cst1(I1,P2,C):
return I1 + P2 < C

```

Here I_1 and P_2 are task and tuning parameters, and C is a global constant variable (see Section 6.1.7 for how to define global constants).

QR EXAMPLE:

```
nodes=1
cores=4
bunit=8 # the block size is an integer multiple of bunit
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 , 16, transform="normalize", name="mb") # row block size (divided by bunit)
nb  = Integer (1 , 16, transform="normalize", name="nb") # column block size (divided by bunit)
lg2npernode  = Integer (0 , int(log2(cores)), transform="normalize", name="lg2npernode") # (log2) of number of MPIs per node
p  = Integer (1 , nodes*cores, transform="normalize", name="p") # number of row processes
PS = Space([mb, nb, lg2npernode, p]) # tuning parameter space
r  = Real (float("-Inf") , float("Inf"), name="r", optimize=True) # runtime
OS = Space([r]) # output space
cst1 = "mb*bunit*p<=m"
cst2 = "nb*bunit*nodes*2**lg2npernode<=n*p"
cst3 = "nodes*2**lg2npernode>=p"
constraints = {"cst1" : cst1, "cst2" : cst2, "cst3" : cst3} # constraints for task parameters and tuning parameters
constants={"nodes":nodes,"cores":cores,"bunit":bunit} # global constants
```

6.1.7 Define the tuning problem with the common autotune interface

```
problem = TuningProblem(IS, PS, OS, objectives, constraints, models, constants)
# define the tuning problem from the spaces, objective function, constraints, cheap performance models and constants. See Section 5 for examples.
```

Input:

- **IS** [Space]: The task parameter space defining the objective function
- **PS** [Space]: The tuning parameter space defining the tuning parameters of the objective function
- **OS** [Space]: The output space defining the output of the objective function
- **objectives** [Callable]: The objective function to be minimized
- **constraints** [Dict]: The constraints for the task parameters and tuning parameters
- **models** [Callable]: The available performance models. This can be None if no model is available.
- **constants** [Dict]: The global constants that can be used in objectives, constraints and models.

Ouput:

- **problem** [Class]: The tuning probleminstance.

6.1.8 Define the computation resource

```
computer = Computer(nodes, cores, hosts) # define the computation resource used in GPTune. See Section 5 for examples.
```

Input:

- **nodes** [Int]: The minimum number of MPI processes used for GPTune's different phases.

- **cores** [Int]: The maximum number of threads per MPI process used for GPTune's different phases.
- **hosts** [Collection]: The list of hostnames. Default to be None.

Output:

- **computer** [Class]: The computer class.

6.1.9 Define and validate the options

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

- **options** [Class] (output): The options class. See GPTune/options.py and Section 6.1.10 for all the options.

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

```
options.validate(computer) # check the options specified by the user, by disabling possibly conflicting features, or reducing the core counts in each phase when the node is oversubscribed.
```

6.1.10 GPTune options

The options affecting the efficiency of GPTune are listed below.

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

    """ Options for GPTune """
    lite_mode = False        # whether to disable all C/C++ and MPI dependencies
    RCI_mode = False         # whether the reverse communication mode will be used
    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 thread per model restart) and search phase (one thread 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
    objective_nospawn = False # Whether the application code is launched via MPI spawn. If True, self['objective_nprocmax'] cores are used per function evaluation, otherwise self['objective_nprocmax']+1 cores are used.

    """ Options for the sampling phase """
    sample_class = 'SampleLHSMDU' # Supported sample classes: 'SampleLHSMDU', 'SampleOpenTURNS'
    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
    sample_random_seed = None # Specify a certain random seed for the pilot sampling phase.

    """ Options for the modeling phase """
    model_class = 'Model_LCM' # Supported sample algorithms: 'Model_GPy_LCM' --LCM from GPy, 'Model_LCM' -- LCM with fast and parallel inversion
    model_kern = 'RBF' # Supported kernels in 'Model_GPy_LCM' model class option -- 'RBF', 'Exponential' or 'Matern12', 'Matern32', 'Matern52'
    model_output_constraint = None # Check output range constraints and disregard out-of-range outputs. Supported options: 'LargeNum': Put a large number, 'Ignore': Ignore those configurations, None: do not check out-of-range outputs.
    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_jitter = 1e-10   # Initial jittering
    model_latent = None # Number of latent functions for building one LCM model, defaults to number of tasks
    model_sparse = False # Whether to use SparseGPRegression or SparseGPCoregionalizedRegression from Model_GPy_LCM
    model_max_jitter_try = 10 # Max number of jittering
    model_random_seed = None # Specify a certain random seed for the surrogate modeling phase


    """ Options for the search phase """
    search_class = 'SearchPyGMO' # Supported searcher classes: 'SearchPyGMO', 'SearchCMO', 'SearchSciPy', 'SearchPyMoo'
    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 algorithms:
```

```
    # 'SearchPyGMO' or 'SearchCMO': 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.
    # 'SearchSciPy': single-objective: 'l-bfgs-b', 'dual_annealing', 'trust-constr', 'shgo'
    # 'SearchPyMoo': single-objective: 'pso' -- particle swarm, 'ga' -- genetic algorithm. multi-objective 'nsga2' -- Non-dominated Sorting GA, 'moead' -- Multi-objective EA vith Decomposition.
    search_pop_size = 1000 # Population size in pgymo or pymoo
    search_gen = 100  # Number of evolution generations in pgymo or pymoo
    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
    search_random_seed = None # Specify a certain random seed for the search phase

    """ Options for transfer learning """
    TLA_method = 'Regression' #"LCM_BF" #'Sum' #'regression_weights_no_scale'
    regression_log_name = 'models_weights.log'

    """ Options for GPTuneBand """
    budget_min = 0.1 # minimum budget
    budget_max = 1 # maximum budget
    budget_base = 2 # the number of arms is floor{log_{budget_base}{budget_max/budget_min}}+1
    fidelity_map = None

    """ Options for cGP """
    N_PILOT_CGP=20      # number of initial samples
    N_SEQUENTIAL_CGP=20 # number of sequential samples
    RND_SEED_CGP=1      # random seed (int)
    EXAMPLE_NAME_CGP='obj_name_dummy' # name of the objective function for logging purpose
    METHOD_CGP='FREQUENTIST' # 'FREQUENTIST' or 'BAYESIAN
    #parameters HMC Bayesian sampling when METHOD_CGP='BAYESIAN'
    N_BURNIN_CGP=500
    N_MCMCSAMPLES_CGP=500
    N_INFERENCE_CGP=500
    EXPLORATION_RATE_CGP=1.0 #Exploration rate is the probability (between 0 and 1) of following the next step produced by acquisition function.
    NO_CLUSTER_CGP=False #If NO_CLUSTER = True, a simple GP will be used.
    N_NEIGHBORS_CGP=3 # number of neighbors for deciding cluster components
    CLUSTER_METHOD_CGP='BGM' #Cluster method: BGM or KMeans
    N_COMPONENTS_CGP=3 # maximal number of clusters
    ACQUISITION_CGP='EI' #acquisition function: EI or MSPE
    BIGVAL_CGP=1e12 #return this big value in the objective function when constraints are not respected

    """ Options for GPTuneHybrid """
    selection_criterion_hybrid='custom' # 'custom' (our novel selection criteria) loglik (log likelihood of the joint GP model), AIC (Akaike information criteria), BIC (Bayesian information criteria), HQC (Hannan-Quinn information criterion).
```

```
    policy_hybrid='UCTS' # MCTS search policy: Currently it supports UCTS (UCB),
    UCTS_var (UCB with variance modification),EXP3 (EXP3),'Multinomial' (Bayesian
    strategy, detailed in our paper).
    exploration_probability_hybrid=0.1 # random search probability when the MCTS
    decides not to roll out a heuristic choice but a random choice.
        n_budget_hybrid=20 # number of total number of samples
        n_pilot_hybrid=10 # number of random pilot samples
        n_find_leaf_hybrid=1 # number of samples in each leaf node.
    acquisition_GP_hybrid='GP-EI' # acuisition functions for the GP of the leaf
    nodes. Currently we support 'GP-EI' and 'GP-UCB'
    random_seed_hybrid=1 # random seed for reproducibility
    bigval_hybrid=1e12 #return this big value in the objective function when
    constraints are not respected
```

Listing 15: Default GPTune options.

6.1.11 Create the data class for storing samples of the spaces

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

Input:

- **problem** [Class]: The tuning problem class.

Output:

- **data** [Class]: The data class to be used for storing sampled tasks, tuning parameters and outputs. This class contains the following properties:
  - data.I=I [list of lists]: Each entry is a list of length α representing one task sample. data.P=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=O [list of numpy arrays]: Each entry is a numpy array of shape $(, \gamma)$ representing the objective function values for one task sample. One row of each array represents the objective function values for one tuning parameter configuration.
  - data.D=D [list of Dicts]: Each entry is a dictionary containing constants/hyperparameters for the data of one task sample.

  On return, data.I=None, data.P=None, data.O=None, data.D=None. But data will follow the above-listed formats once the tuning is completed. See Section 6.2.4 for details.

6.2 Calling different tuning algorithms

6.2.1 Initialize GPTune (needed by MLA, TLA_I and TLA_II)

```
gptune = GPTune(problem, computer, data, historydb, options, driverabspath,
    models_update)
# initialize the tuner from the meta data
```

Input:

- **problem** [Class]: The tuning problem
- **computer** [Class]: The computer
- **data** [Class]: The data class. If any of data.I, data.P and data.O is None, the tuner will generate random samples for it later.
- **historydb** [Class]: The historydb class. If None, historydb=HistoryDB() will be used to create a historydb class.
- **options** [Class]: The options class.
- **driverabspath** [string]: Absolute path of the file containing this call. Default to None. We recommend passing driverabspath=os.path.abspath(__file__).
- **models_update** [Callable]: The cheap performance update function (see Section 6.1.4). This can be set to None if there is no performance model, or the performance model requires no update.

Output:

- **gptune** [Class]: The tuner class that registers problem, computer, data and options as gptune.problem, gptune.computer, gptune.data, and gptune.options, respectively.

6.2.2 Evaluate with manually given parameter configurations

The user can optionally invoke this function if the user wants to start evaluating with manually given parameter samples for a given tuning task.

```
data = gptune.EvaluateObjective(Tgiven, Pgiven)
```

- **gptune** [Class]: The tuning class instance initialized from Section 6.2.1.
- **EvaluateObjective** [Class method]: The class method to evaluate for manually given parameter configurations.

Input:

- **Tgiven** [list]: A list of the task parameters of the given single task.
- **Pgiven** [list of lists]: A list of the lists of the tuning parameters to be evaluated.

Output:

- **data** [Class]: 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.

6.2.3 Call single-task learning algorithm (standard GP)

Call the Bayesian optimization-based autotuning using GP surrogate modeling for a given single task.

```
(data, models, stats) = gptune.SLA(NS, NS1, Tgiven)
```

- **gptune** [Class]: The tuning class instance initialized from Section 6.2.1.
- **SLA** [Class method]: The class method to run SLA.

Input:

- **NS** [Int]: Number of total samples per task to be returned. Note that the tuner returns immediately if the number of samples in the historical data is more than NS.
- **NS1** [Int]: If no historical data is present, the tuner generates NS1 initial random tuning parameter samples before starting the adaptive model refinement.
- **Tgiven** [list]: A list of prescribed task parameters of the given single task. Note that Tgiven should match the list of task parameters in the historical data (if present).

Output:

- **data** [Class]: 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]: Each entry represents the trained LCM model for one objective in the adaptive model refinement.
- **stats** [Dict]: Stats of the autotuning (e.g., runtime taken by each modeling, search, and function evaluation step).

6.2.4 Call multi-task learning algorithm (MLA)

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

- **gptune** [Class]: The tuning class instance initialized from Section 6.2.1.
- **MLA** [Class method]: The class method to run MLA.

Input:

- **NS** [Int]: Number of total samples per task to be returned. Note that the tuner returns immediately if the number of samples in the historical data is more than NS.
- **NS1** [Int]: If no historical data is present, the tuner generates NS1 initial random tuning parameter samples before starting the adaptive model refinement.
- **NI** [Int]: Number of tasks to be modeled (i.e., NI=n_i). If one needs to add new tasks to the historical data, use the TLA interface in Section 6.2.6 instead.

- **Tgiven** [list of lists]: A list of prescribed task parameters. Note that Tgiven should match the list of task parameters in the historical data (if present).

Output:

- **data** [Class]: 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]: Each entry represents the trained LCM model for one objective in the adaptive model refinement.
- **stats** [Dict]: Stats of the autotuning (e.g., runtime taken by each modeling, search, and function evaluation step).

6.2.5 Call transfer learning algorithm (TLA Type I)

As explained as the first type of algorithm in Section 3.1.4, users can run transfer learning with the ability to load surrogate model functions. This is to build a surrogate model for the new task leveraging performance data for existing tasks. The following interface builds one or several surrogate models for the target task depending on the options['TLA_method'].

```
# build the MLA models and search for the optimum tuning parameters of each task
(data, models,stats) = gptune.TLA_I(NS, Tnew, models_transfer,
    source_function_evaluations, TLA_options)
```

- **gptune** [Class]: The tuning class instance initialized from Section 6.2.1.
- **TLA** [Class method]: The class method to run TLA.

Input:

- **NS** [Int]: Number of total samples per task to be returned. Note that the tuner returns immediately if the number of samples in the historical data is more than NS.
- **Tnew** [list of lists]: A list of prescribed task parameters describing the target task. Note that Tgiven should match the list of task parameters in the historical data (if present).
- **models_transfer** [list of functions]: A list of the source surrogate model functions to be utilized for transfer learning. Each surrogate model function can be obtained through the database interface described in Section 6.3.4.
- **source_function_evaluations** [list of list of Dict]: The dictionary storing the performance data of the source tasks.
- **TLA_options** [list of strings]: A list of TLA approaches to be considered in the ensemble-based TLA.

Output:

- **data** [Class]: 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]: Each entry represents the trained LCM model for one objective in the adaptive model refinement.
- **stats** [Dict]: Stats of the autotuning (e.g., runtime taken by each modeling, search, and function evaluation step).

As discussed in Section 6.2.5, we provide several different methods to learn knowledge from the given surrogate models. For example, GPTune supports an LCM-based modeling to understand the dependence between different tasks, or a summation of multiple surrogate models with some different weighting approaches. Please refer to Section 6.1.10 for the supported TLA options.

6.2.6 Call transfer learning algorithm (TLA Type II)

As explained as the second type of algorithm in Section 3.1.4, users can predict the optimal parameter of the new task without collecting any data by using existing data. The goal is to predict the optimal parameter of the new task using optimal parameters for existing tasks. The following interface builds a model that directly predicts the optimal tuning parameters of the new task requesting only function evaluation data of existing tasks [37].

```
(aprxopts,objval,stats) = gptune.TLA_II(Tnew, Tsrc, source_function_evaluations)
# use existing data on pre-tuned tasks (stored in gptune.data) and tla to search
    for the optimum tuning parameters of each new task
```

- **gptune** [Class]: The tuning class instance initialized from Section 6.2.1.
- **TLA** [Class method]: The class method to run TLA.

Input:

- **Tnew** [list of lists] (input): Tnew consists of a list of prescribed tasks to be tuned
- **Tsrc** [list of lists] (input): list of the task parameters of the source tasks.
- **source_function_evaluations** [list of list of Dict] (input): The dictionary storing the performance data of the source tasks.

Output:

- **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

6.2.7 Call OpenTuner

```
from callopentuner import OpenTuner
(data,stats)=OpenTuner(T, NS, tp, computer, run_id)
# initialize and call OpenTuner to generate objective function samples. If there
    are multiple tasks, OpenTuner is invoked one task each time.
```

Input:

- **T** [list of lists]: A list of prescribed task parameters of length NI.
- **NS** [Int]: Number of total samples per task to be returned.
- **tp** [Class]: The tuning problem defined in Section 6.1.7.
- **computer** [Class]: The computer.
- **runid** [String]: Name of the tuner (default to "OpenTuner").

Output:

- **data** [Class]: The data class containing all the task, tuning parameters and outputs sampled by the tuner. len(data.I)=NI, len(data.P)=NI, len(data.O)=NI, and len(data.P[0])=NS.
- **stats** [Dict]: Memory and CPU profiles generated by the tuner

6.2.8 Call HpBandSter (TPE)

```
from callhpbandster import HpBandSter
(data,stats)=HpBandSter(T, NS, tp, computer, run_id)
# initialize and call OpenTuner to generate objective function samples. If there
    are multiple tasks, OpenTuner is invoked one task each time.
```

Input:

- **T** [list of lists]: A list of prescribed task parameters of length NI.
- **NS** [Int]: Number of total samples per task to be returned.
- **tp** [Class]: The tuning problem defined in Section 6.1.7.
- **computer** [Class]: The computer.
- **runid** [String]: Name of the tuner (default to "HpBandSter").

Output:

- **data** [Class]: The data class containing all the task, tuning parameters and outputs sampled by the tuner. len(data.I)=NI, len(data.P)=NI, len(data.O)=NI, and len(data.P[0])=NS.
- **stats** [Dict]: Memory and CPU profiles generated by the tuner

6.2.9 Call HpBandSter (Multi-fidelity)

```
from callhpbandster import HpBandSter_bandit
(data,stats)=HpBandSter_bandit(T, NS, tp, computer, run_id)
# initialize and call HpBandSter to generate objective function samples. If there
    are multiple tasks, HpBanSter is invoked one task each time.
```

Input:

- **T** [list of lists]: A list of prescribed task parameters of length NI.
- **NS** [Int]: Number of total samples per task to be returned.
- **tp** [Class]: The tuning problem defined in Section 6.1.7.
- **computer** [Class]: The computer.
- **runid** [String]: Name of the tuner (default to "hpbandster_bandit").

Output:

- **data** [Class]: The data class containing all the task, tuning parameters and outputs sampled by the tuner. len(data.I)=NI, len(data.P)=NI, len(data.O)=NI, and len(data.P[0])=NS.
- **stats** [Dict]: Memory and CPU profiles generated by the tuner

6.2.10 Call cGP

```
from callcgp import cGP
(data,stats)=cGP(T, tp, computer, options, run_id)
# initialize and call cGP to generate objective function samples. If there are
    multiple tasks, cGP is invoked one task each time.
```

Input:

- **T** [list of lists]: A list of prescribed task parameters of length NI.
- **options** [Class]: The GPTune Options class. The following options need to be set: options['EXAMPLE_NAME_CGP']=[APP_NAME], options['N_PILOT_CGP']=NS1, options['N_SEQUENTIAL_CGP']=NS-options['N_PILOT_CGP']. See Section 6.1.10 for more details.
- **tp** [Class]: The tuning problem defined in Section 6.1.7.
- **computer** [Class]: The computer.
- **runid** [String]: Name of the tuner (default to "cGP").

Output:

- **data** [Class]: The data class containing all the task, tuning parameters and outputs sampled by the tuner. len(data.I)=NI, len(data.P)=NI, len(data.O)=NI, and len(data.P[0])=NS.
- **stats** [Dict]: Memory and CPU profiles generated by the tuner

6.2.11 Call GPTuneHybrid (hybridMinimization)

```
from callhybrid import GPTuneHybrid
(data,stats) = GPTuneHybrid(T, tp, computer, options, run_id)
# initialize and call GPTuneHybrid to generate objective function samples. If
    there are multiple tasks, GPTuneHybrid is invoked one task each time.
```

Input:

- **T** [list of lists]: A list of prescribed task parameters of length NI.
- **options** [Class]: The GPTune Options class. The following options need to be set such that options['n_pilot_hybrid']=NS1, options['n_find_tree_hybrid']*options['n_find_leaf_hybrid']=NS-NS1. See Section 6.1.10 for more details.
- **tp** [Class] (output): The tuning problem defined in Section 6.1.7.
- **computer** [Class]: The computer.
- **runid** [String]: Name of the tuner (default to "GPTuneHybrid").

Output:

- **data** [Class]: The data class containing all the task, tuning parameters and outputs sampled by the tuner. len(data.I)=NI, len(data.P)=NI, len(data.O)=NI, and len(data.P[0])=NS.
- **stats** [Dict]: Memory and CPU profiles generated by the tuner

6.2.12 Call GPTuneBand (Multi-fidelity)

Initialize the GPTuneBand class:

```
gptune = GPTune_MB(problem, computer, options)
# initialize the GPTune_MB tuner from the meta data
```

Input:

- **problem** [Class]: The tuning problem.
- **computer** [Class]: The computer.
- **options** [Class]: The options class.

Output:

- **gptune** [Class]: The tuner class that registers problem, computer, data and options as gptune.problem, gptune.computer, gptune.data, and gptune.options, respectively.

Invoke the GPTuneBand tuning:

```
(data, stats, data_hist) = gptune.MB_LCM(NLOOP, Tgiven, Pdefault)
# build a multi-fidelity, multi-task LCM model and search for the optimum tuning
    parameters of each task
```

Input:

- **NLOOP** [Integer]: Number of passes executing GPTuneBand, see [41] for more details.
- **Tgiven** [list of lists]: A list of prescribed task parameters. Note that Tgiven should match the list of task parameters in the historical data (if present).
- **Pdefault** [List]: A list specifying the first sample (tuning parameter configuration) for all tasks.

Output:

- **data** [Class]: 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. Note that the data only has the highest-fidelity function evaluations.
- **stats** [Dict]: Memory and CPU profiles generated by the tuner.
- **data_hist** [Class]: The data class containing the samples drawn by GP for all tasks and fidelities (i.e., excluding the samples from successive halving).

6.3 Utility functions enabled by the history database

GPTune provides a number of useful utility functions (in Python) that help analyze with collected functions evaluations in the GPTune database. Many of those utility functions require dictionary inputs, called `"problem_space"` and `"configuration_space"`. `"problem_space"` describes the task, tuning, and output parameter space and constraints to be considered for analysis the utility function. `"configuration_space"` describes the machine and software runtime environments to be considered for analysis. This section first explains how to describe the `"problem_space"` and `"configuration_space"`, and then introduces the utility functions.

6.3.1 Describing Problem Space

```
problem_space = {
    "input_space": [
        {"name":"t", "type":"real", "transformer":"normalize",
         "lower_bound":1.0, "upper_bound":10.0}
    ],
    "parameter_space": [
        {"name":"x", "type":"real", "transformer":"normalize",
         "lower_bound":0.0, "upper_bound":1.0}
    ],
    "output_space": [
        {"name":"y", "type":"real", "transformer":"identity",
         "lower_bound":float('-Inf'), "upper_bound":float('Inf')}
    ],
    "constants": [
        {"c1": 1, "c2": 2},
        {"c1": 10, "c2": 20}
    ]
}
```

- `input_space` [list of Dicts] (input): list of the input parameter spaces to be considered for a utility function.

- `parameter_space` [list of Dicts] (input): list of the tuning parameter spaces to be considered for a utility function.
- `output_space` [list of Dicts] (input): list of the output spaces to be considered for a utility function.
- `constants` [list of Dicts] (optional input): list of the constant variable sets to be considered for a utility function.

6.3.2 Describing configuration space

```
configuration_space = {
    "machine_configurations": [
        { "Cori": { "haswell": { "nodes": 1, "cores": 32 }}}
    ],
    "software_configurations": [
        { "gcc": { "version_from": [8,0,0], "version_to":[9,0,0] }}
    ],
    "user_configurations": [
        "user_A", "user_B", "user_C"
    ]
}
```

- `machine_configurations` [list of Dicts] (optional input): list of the machine configurations to be considered for a utility function. If not provided, the utility function will use all the data without distinguishing data using their machine configuration.
- `software_configurations` [list of Dicts] (optional input): list of the software configurations to be considered for a utility function. If not provided, the utility function will use all the data without distinguishing data using their software configuration.
- `user_configurations` [list of Dicts] (optional input): list of the users to be considered for a utility function. If not provided, the utility function will use all the data regardless of the user information of performance data.

6.3.3 Query Function Evaluations from the Local Database

```
import gptune
func_evals = gptune.QueryFunctionEvaluations(problem_space, historydb_path)
```

Input:

- **problem_space** [Dict]: The dictionary specifying the requested task parameter space, tuning parameter space, output space and constants. If None, it will not check on the tuning space and consider all the data in the database.
- **configuration_space** [Dict]: The dictionary specifying the machine and software environment to be considered. If None, it will not check on the environment information and consider all the data in the database.
- **historydb_path** [String]: Path to the history database file.

Output:

- **func_evals** [list of Dict]: List of function evaluations (dictionaries).

6.3.4 Build a Surrogate Model from the Local Database

The following interfaces take performance data in the database and build a surrogate performance model. A performance model generated through these interfaces is a callable black-box function that takes the task and parameter information using a Python Dict and returns the predicted values by the surrogate model. For example, such a generated function is given as follows (model_function is the returned surrogate model from the interfaces):

```
# a model function for PDGEQRF routine whose task parameters are m and n and
    tuning parameters are mb, nb, lg2npernode, and p.
ret = model_function({"m": 4000, "n": 2000, "mb": 16, "nb": 16,
"lg2npernode": 5, "p": 13})}
# output is also a dictionary e.g., { "r": 0000 }
print (ret)
```

This feature can be used for transfer learning and sensitivity analysis, discussed in Section 6.2.5 and 3.1.8.

Build a surrogate model from function evaluation data Users can build a surrogate performance model using function evaluation data in the database. The user can manually read the database file and pass the list of function evaluations to this interface.

Users can also leverage the advanced history database API to query function evaluation data more efficiently, e.g., directly downloading from the crowd-tuning repository depending on a certain query command. Please use Section 6.4.4 for the database API.

```
# Build a surrogate model from function evaluation data
(model_function) = gptune.BuildSurrogateModel(problem_space, modeler, input_task,
    function_evaluations)
```

- **problem_space** [Dict] (input): The dictionary specifying the requested task parameter space, tuning parameter space, output space and constants.
- **modeler** [string]: The modeler to fit the downloaded function evaluation data. Currently, it can be "Model_GPy_LCM" or "Model_LCM".
- **input_task** [list] (input): The list defining the target task.
- **function_evaluations** [list of Dict] (input): list of function evaluation data. Each function evaluation is described in the JSON format in Section 3.3.4.
- **model_function** [Callable] (output): The function that returns the mean and variance with the task and tuning paramters specified by an input dictionary.

Read a surrogate model from the database Instead of re-training performance models using function evaluation data, users can read the metadata (i.e. hyperparameters of the trained model) and reproduce the same surrogate model. The following interface takes the database record of an MLA surrogate model as an input, reproduces the surrogate model from the record data, and returns the reproduced surrogate model. Refer to Section 3.3.4 for the details of the JSON database format that stores an MLA performance model.

```
import gptune
model_function = gptune.ReadSurrogateModel(problem_space, configuration_space,
    historydb_path, method)
# Choose a surrogate model from the database and returns the model black-box
    function.
```

- **problem_space** [Dict] (input): The dictionary specifying the requested task parameter space, tuning parameter space, output space and constants. If None, it will not check on the tuning space and consider all the data in the database.
- **configuration_space** [Dict] (input): The dictionary specifying the machine and software environment to be considered. If None, it will not check on the environment information and consider all the data in the database.
- **method** [String] (Optional Input): Model selection criterion to select one model from multiple loadable models. ***Parameters.*** "max_evals" (default): choose the model that has most function evaluation data. "MLE" or "mle": choose the model that has the highest likelihood. "AIC" or "aic": choose the model based on Akaike Information Criterion (AIC). "BIC" or "bic": choose the model based on Bayesian Information Criterion (BIC).
- **model_function** [Function] (output): A callable Python function (from the user side) that takes the task and parameter information using a Python Dict and returns the predicted values by the surrogate model for the given task and parameter configuration.

6.3.5 Predict Output of a Given Tuning Parameter Configuration

```
import gptune
ret = gptune.PredictOutput(problem_space, modeler, input_task, input_parameter,
    surrogate_model, function_evaluations)
```

- **problem_space** [Dict] (input): The dictionary specifying the requested task parameter space, tuning parameter space, output space and constants.
- **modeler** [string]: The modeler to fit the downloaded function evaluation data. Currently, it can be "Model_GPy_LCM" or "Model_LCM". Default to "Model_GPy_LCM".
- **input_task** [list] (input): The list defining the target task.
- **input_parameter** [list] (input): The list defining the tuning parameters to be predicted.
- **surrogate_model** [Callable] (input): A callable Python function that takes the task and parameter information using a Python Dict and returns the predicted values by the surrogate model for the given task and parameter configuration. If None, a surrogate model will be created using BuildSurrogateModel in Section 6.3.4.

- **function_evaluations** [list of Dict] (input): The dictionary storing the performance data.
- **ret** [Dict] (output): A dictionary containing the predicted mean and variance.
  - ret["mu"] = predicted mean value.
  - ret["var"] = variance of the prediction.

6.3.6 Sensitivity Analysis

The GPTune interface for the sensitivity analysis in Section 3.1.8 is given by:

```
(Si) = SensitivityAnalysis(problem_space, modeler, method, input_task,
    surrogate_model, function_evaluations, historydb_path, tuning_problem_name,
    num_samples)
# Reproduce the MLA surrogate model from a database record.
```

Input:

- **problem_space** [Dict] (input): The dictionary specifying the requested task parameter space, tuning parameter space, output space and constants.
- **modeler** [string]: The modeler to fit the downloaded function evaluation data. Currently, it can be "Model_GPy_LCM" or "Model_LCM".
- **method** [String] (Input): Sensitivity analysis algorithm. Default to "Sobol".
- **input_task** [list] (input): The list defining the target task.
- **surrogate_model** [Callable] (input): A callable Python function that takes the task and parameter information using a Python Dict and returns the predicted values by the surrogate model for the given task and parameter configuration. If None, a surrogate model will be created using BuildSurrogateModel in Section 6.3.4.
- **function_evaluations** [Dict] (input): The dictionary storing the performance data.
- **historydb_path** [String] (input): If function_evaluations is None, the function will try to load function evaluation data from historydb_path.
- **tuning_problem_name** [String] (input): If both function_evaluations and historydb_path are None, the function will try to load function evaluation data from "gptune.db/"+tuning_problem_name+".json".
- **num_samples** [Integer] (input): How many samples (from the surrogate model) are generated and used to run the Sobol analysis.

Output:

- **Si** [Dict] (output): Sobol analysis output from SALib. Si is a Python dict with the keys "S1", "S2", "ST", "S1_conf", "S2_conf", and "ST_conf". The _conf keys store the corresponding confidence intervals, with a confidence level of 95% The S1, S2, and ST represents first-order, second-order, and total-order sensitivity indices; the detailed descriptions are given as follows.

6.4 Crowd-tuning API: using shared database

6.4.1 Crowd-tuning API: Query performance data from the shared database

```
import crowdtune
function_evaluations = crowdtune.QueryFunctionEvaluations(api_key,
    tuning_problem_name, problem_space, configuration_space)
# query the shared database using the API request
```

Input:

- **api_key** [string]: The personal API key to access the shared database (see Section 3.4.3 for the details about API keys).
- **tuning_problem_name** [string]: The application name (one of the parameters) for the API request.
- **problem_space** [Dict]: The dictionary specifying the requested task parameter space, tuning parameter space, output space and constants. If None, it will not check on the tuning space and consider all the data in the database. Section 6.3.1 explains how to describe a `problem_space` dictionary.
- **configuration_space** [Dict]: The dictionary specifying the machine and software environment to be considered. If None, it will not check on the environment information and consider all the data in the database. Section 6.3.2 explains how to describe a `configuration_space` dictionary.

Output:

- **function_evaluations** [list of Dict]: List of the dictionaries storing the queried function evaluation results.

6.4.2 Crowd-tuning API: Upload performance data to the shared database

```
import crowdtune
r = crowdtune.UploadFunctionEvaluation(api_key, function_evaluation)
# upload a function evaluation result to the shared database
```

Input:

- **api_key** [string]: The personal API key to access the shared database (see Section 3.4.3 for the details about API keys).
- **function_evaluation** [Dict]: The dictionary of the function evaluation to be uploaded (see Section 3.3.4 for the JSON format).

Output:

- **r** [integer]: The return code (`r==200` means the query is success).

6.4.3 Crowd-tuning API: Make a prediction using the crowd repository

```
import crowdtune
prediction = crowdtune.QueryPredictOutput(api_key, tuning_problem_name,
    problem_space, configuration_space, modeler, input_task, input_parameter,
    objective_name)
# query the shared database to make a function prediction
```

Input:

- **api_key** [string]: The personal API key to access the shared database (see Section 3.4.3 for the details about API keys).
- **tuning_problem_name** [string]: The application name (one of the parameters) for the API request.
- **problem_space** [Dict]: The dictionary specifying the requested task parameter space, tuning parameter space, output space and constants. If None, it will not check on the tuning space and consider all the data in the database. Section 6.3.1 explains how to describe a `problem_space` dictionary.
- **configuration_space** [Dict]: The dictionary specifying the machine and software environment to be considered. If None, it will not check on the environment information and consider all the data in the database. Section 6.3.2 explains how to describe a `configuration_space` dictionary.
- **input_task** [list]: The list defining the target task.
- **input_parameter** [Dict or list]: The list (or dictionary) defining the tuning parameter configuration of interest.
- **modeler** [string]: The modeler to fit the downloaded function evaluation data. Currently, it can be "Model_GPy_LCM" or "Model_LCM".
- **objective_name** [string]: The name of the objective (in case of multiple objectives) to be predicted.

Output:

- **prediction** [Dict]: A dictionary containing the predicted mean and variance.

6.4.4 Crowd-tuning API: Query a surrogate performance model

```
import crowdtune
surrogate = crowdtune.QuerySurrogateModel(api_key, tuning_problem_name,
    problem_space, configuration_space, modeler, input_task)
# query the shared database to generate a surrogate model that can output the mean
    and variance of a given task
```

Input:

- **api_key** [string]: The personal API key to access the shared database (see Section 3.4.3 for the details about API keys).

- **tuning_problem_name** [string]: The application name (one of the parameters) for the API request.
- **problem_space** [Dict]: The dictionary specifying the requested task parameter space, tuning parameter space, output space and constants. If None, it will not check on the tuning space and consider all the data in the database. Section 6.3.1 explains how to describe a `problem_space` dictionary.
- **configuration_space** [Dict]: The dictionary specifying the machine and software environment to be considered. If None, it will not check on the environment information and consider all the data in the database. Section 6.3.2 explains how to describe a `configuration_space` dictionary.
- **input_task** [list]: The list defining the target task.
- **input_parameter** [Dict or list]: The list (or dictionary) defining the tuning parameter configuration of interest.
- **modeler** [string]: The modeler to fit the downloaded function evaluation data. Currently, it can be "Model_GPy_LCM" or "Model_LCM".

Output:

- **surrogate** [Callable]: A callable that can estimate the mean and variance for the task input_task on any tuning parameter configuration.

6.4.5 Crowd-tuning API: Run a sensitivity analysis (SA)

```
import crowdtune
Si = crowdtune.QuerySensitivityAnalysis(api_key, tuning_problem_name,
    problem_space, configuration_space, modeler, method, input_task, num_samples)
# query the shared database to perform a sensitivity analysis can output the mean
    and variance of a given task
```

Input:

- **api_key** [string]: The personal API key to access the shared database (see Section 3.4.3 for the details about API keys).
- **tuning_problem_name** [string]: The application name (one of the parameters) for the API request.
- **problem_space** [Dict]: The dictionary specifying the requested task parameter space, tuning parameter space, output space and constants. If None, it will not check on the tuning space and consider all the data in the database. Section 6.3.1 explains how to describe a `problem_space` dictionary.
- **configuration_space** [Dict]: The dictionary specifying the machine and software environment to be considered. If None, it will not check on the environment information and consider all the data in the database. Section 6.3.2 explains how to describe a `configuration_space` dictionary.

- **input_task** [list]: The list defining the target task.
- **modeler** [string]: The modeler to fit the downloaded function evaluation data. Currently, it can be "Model_GPy_LCM" or "Model_LCM".

Output:

- **Si** [Dict]: Sobol analysis output from SALib. Si is a Python dict with the keys "S1", "S2", "ST", "S1_conf", "S2_conf", and "ST_conf". The _conf keys store the corresponding confidence intervals, typically with a confidence level of 95%. The S1, S2, and ST represents first-order, second-order, and total-order sensitivity indices.

6.5 Command line invocations

6.5.1 Invoke GPTune in default mode

```
mpirun --oversubscribe --allow-run-as-root --mca pmix_server_max_wait 3600 --mca
    pmix_base_exchange_timeout 3600 --mca orte_abort_timeout 3600 --mca
    plm_rsh_no_tree_spawn true -n 1  python ./application_tuner_driver.py
```

Here application_tuner_driver.py is the GPTune Python driver that contains definitions from Section 6.1.2 to Section 6.2.12. The above command invokes the GPTune tuning process. Note that these MPI runtime parameters are necessary for OpenMPI 4.0.1, higher versions have not been tested extensively.

6.5.2 Invoke GPTune in lite mode

```
export GPTUNE_LITE_MODE=1
python ./application_tuner_driver.py
```

Here application_tuner_driver.py is the GPTune Python driver that contains definitions from Section 6.1.2 to Section 6.2.12, the same as Section 6.5.1. Note that the lite mode does not require mpi4py, and MPI spawning feature of GPTune can be supported. See Section 3.2 for more details.

6.5.3 Invoke GPTune in RCI mode

```
python ./application_tuner_driver_rci.py
```

Here application_tuner_driver_rci.py is the GPTune Python driver similar to application_tuner_driver.py, which contains definitions from Section 6.1.3 to Section 6.2.12. For RCI mode, the above command will execute phases in GPTune as usual, but without calling the application code. Instead, when function evaluation is needed, the required samples will be stored in the database ./gptune_db/application_name.json, which is located in the same directory as application_tuner_driver_rci.py. The user can then search for the required samples in the database, invoke the application code in bash, write the evaluation results into the database, and then call the GPTune Python driver again to ask for the next samples. As such, there is no need to define the objective function as in Section 6.1.2 or modify the application code as in Section 6.1.1. Moreover, OpenMPI and its runtime parameters are not mandatory (e.g., MPICH, Spectrum MPI, or Intel MPI can also be used). Note that in application_tuner_driver_rci.py, options['RCI_mode']=True is required. In addition, the interface to OpenTuner (Section 6.2.7) or HpBandSter (Section 6.2.8) is not supported in RCI mode. See Section 4.3 for a complete example.

7 Known Issues and Solutions

This section collects a few known installation and runtime issues and their solutions. Also see `https://github.com/gptune/GPTune/tree/master/FAQ`.

Installation issue: multiple Python and pip versions. If one's system has multiple versions of Python and pip, one needs to make sure the same version of Python and pip is used to install GPTune. One can consider using the virtual environment venv to install GPTune and its Python dependencies to a user-specified directory.

```
alias python=path-to-your-python-executable
alias pip=path-to-your-pip-executable
python -m pip install virtualenv
rm -rf myenv
python -m venv myenv
source myenv/bin/activate
unalias pip  # this makes sure virtualenv install packages at its own site-
    packages directory
unalias python

```

Then all packages you installed afterwards will be put into ./myenv/lib/pythonX.X/site-packages.

Cori: runtime error: "Error in 'python': break adjusted to free malloc space: 0x000001 0000000000 *"** One needs to unload the craype-hugepages2M module with: module unload craype-hugepages2M

Cori: runtime error: "_pmi_alps_init:alps_get_placement_info returned with error -1" Intead of "python xx.py", one needs "srun/mpirun -n 1 xx.py" to get the correct MPI infrastructure.

Runtime: hanging at "MLA iteration: 0" This typically means that a wrong version of openmpi is used at runtime. Make sure to use the same openmpi version as the one for gptune installation.

Runtime error: "ImportError: cannot import name '_centered' from 'scipy.signal.signal tools'" This is due to the use of scipy-1.8.0 (or later) and statsmodels-0.12.2 (or earlier). You can upgrade statsmodels to 0.13.2.

Installation error: "lto1: internal compiler error: invalid line in the resolution file" We noticed this build error for Thread Building Block (TBB) on Summit due to the compiler flag '-flto' from the interprocedural optimization (IPO). One can disable that by passing -DTBB_ENABLE_IPO=OFF to the CMake build of TBB.

Installation error: "Could not find a version that satisfies the requirement pygmo (from versions: none)" For Python3.9+, "pip install pygmo" does not work. You need to install pygmo from source. Assume that your system has BOOST(>=1.69) installed at 'BOOST_ROOT' and pybind11 installed at 'SITE_PACKAGE'/pybind11. The C and C++ compilers are 'MPICC' and 'MPICXX'. If you do not have BOOST>=1.69, you can install it from source:

```
cd $GPTUNEROOT
wget -c 'http://sourceforge.net/projects/boost/files/boost/1.69.0/boost_1_69_0.tar.bz2/download'
tar -xvf download
cd boost_1_69_0/
./bootstrap.sh --prefix=$PWD/build
./b2 install
export BOOST_ROOT=$GPTUNEROOT/boost_1_69_0/build
```

The following lines install TBB, pagmo and pygmo from source:

```
export TBB_ROOT=$GPTUNEROOT/oneTBB/build
export pybind11_DIR=$SITE_PACKAGE/pybind11/share/cmake/pybind11
export BOOST_ROOT=XXX  # if insalled from source, remove this line
export pagmo_DIR=$GPTUNEROOT/pagmo2/build/lib/cmake/pagmo

cd $GPTUNEROOT
rm -rf oneTBB
git clone https://github.com/oneapi-src/oneTBB.git
cd oneTBB
mkdir build
cd build
cmake ../ -DCMAKE_INSTALL_PREFIX=$PWD -DCMAKE_INSTALL_LIBDIR=$PWD/lib -DCMAKE_C_COMPILER=$MPICC -DCMAKE_CXX_COMPILER=$MPICXX -DCMAKE_VERBOSE_MAKEFILE:BOOL=ON
make -j16
make install
git clone https://github.com/wjakob/tbb.git
cp tbb/include/tbb/tbb_stddef.h include/tbb/.

cd $GPTUNEROOT
rm -rf pagmo2
git clone https://github.com/esa/pagmo2.git
cd pagmo2
mkdir build
cd build
cmake ../ -DCMAKE_INSTALL_PREFIX=$PWD -DCMAKE_C_COMPILER=$MPICC -DCMAKE_CXX_COMPILER=$MPICXX -DCMAKE_INSTALL_LIBDIR=$PWD/lib
make -j16
make install
cp lib/cmake/pagmo/*.cmake .

cd $GPTUNEROOT
rm -rf pygmo2
git clone https://github.com/esa/pygmo2.git
cd pygmo2
mkdir build
cd build
cmake ../ -DCMAKE_INSTALL_PREFIX=$PREFIX_PATH -DPYGMO_INSTALL_PATH="$SITE_PACKAGE" -DCMAKE_C_COMPILER=$MPICC -DCMAKE_CXX_COMPILER=$MPICXX
make -j16
make install
```

Note that, if you have installed TBB, PaGMO, and PyGMO2 from source codes with the installation paths described in the above instructions, then, to use the PyGMO module ("import pygmo") in GPTune, you may need to provide the path to the TBB and PaGMO library, for example:

```
export LD_LIBRARY_PATH=GPTUNEROOT/pagmo2/build/lib:LD_LIBRARY_PATH
export LD_LIBRARY_PATH=GPTUNEROOT/oneTBB/build/lib:LD_LIBRARY_PATH
```

Runtime error: "ModuleNotFoundError: No module named 'fn'" This is due to the dependency on fn from OpenTuner. Note that fn has been removed from opentuner≥0.8.8, which should avoid this runtime error.

Runtime error: "module openturns has no module RandomSample" This is typically due to the fact that OpenTURNs installation is incomplete as it requires TBB, which only works on AMD and Intel-based architectures. You can switch OpenTURNs off by setting options['sample_class'] = 'SampleLHSMDU'.

8 Acknowledgements

The code is available at: `http://github.com/GPTune`. Please cite this user guide and software as the following bibTex item:

```
@misc{gptuneUser,
title={GPtune User Guide},
author={Cho, Younghyun and Demmel, James W. and Dinh, Grace and Li, Xiaoye S.
and Liu, Yang and Luo, Hengrui and Marques, Osni and Sid-Lakhdar, Wissam M.},
year={2020-}
}
```

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.

We acknowledge following collaborators and colleagues for their helpful and constructive comments and feedback during this research development: Riley J. Murray (UC Berkeley) and Rahul Jain (UC Berkeley).

Alphabetical Index

References

[1] J. Bergstra, R. Bardenet, Y. Bengio, and B. Kégl. Algorithms for Hyper-Parameter Optimization. *Advances in neural information processing systems*, 24, 2011.

[2] F. Biscani and D. Izzo. A parallel global multiobjective framework for optimization: pagmo. *Journal of Open Source Software*, 5(53):2338, 2020.

[3] 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.

[4] J. Blank and K. Deb. Pymoo: Multi-Objective Optimization in Python. *IEEE Access*, 8:89497–89509, 2020.

[5] Y. Cho, J. W. Demmel, X. S. Li, Y. Liu, and H. Luo. Enhancing Autotuning Capability with a History Database. In *The IEEE 14th International Symposium on Embedded Multicore/-Manycore SoCs (MCSoC-2021)*, volume 20, 2021.

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

[7] L. Dalcín, R. Paz, and M. Storti. MPI for Python. *Journal of Parallel and Distributed Computing*, 65(9):1108–1115, 2005.

[8] 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.

[9] E. Dolstra and A. Löh. Nixos: A purely functional linux distribution. In *Proceedings of the 13th ACM SIGPLAN international conference on Functional programming*, pages 367–378, 2008.

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

[11] S. Falkner, A. Klein, and F. Hutter. BOHB: Robust and Efficient Hyperparameter Optimization at Scale. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 1437–1446. PMLR, 10–15 Jul 2018.

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

[13] T. Gamblin, M. LeGendre, M. R. Collette, G. L. Lee, A. Moody, B. R. de Supinski, and S. Futral. The Spack Package Manager: Bringing Order to HPC Software Chaos. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, SC '15, New York, NY, USA, 2015. Association for Computing Machinery.

[14] E. C. Garrido-Merchán and D. Hernández-Lobato. Dealing with categorical and integer-valued variables in Bayesian Optimization with Gaussian processes. *Neurocomputing*, 380:20–35, 2020.

[15] P. Ghysels, G. Chávez, L. Guo, C. Gorman, X. S. Li, Y. Liu, L. Rebrova, F.-H. Rouet, T. Mary, and J. Actor. STRUMPACK, 2016-.

[16] D. Golovin, B. Solnik, S. Moitra, G. Kochanski, J. Karro, and D. Sculley. Google Vizier: A Service for Black-Box Optimization. In *Proceedings of the 23rd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '17, page 1487–1495, New York, NY, USA, 2017. Association for Computing Machinery.

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

[18] T. Head, M. Kumar, H. Nahrstaedt, G. Louppe, and I. Shcherbatyi. scikit-optimize/scikit-optimize. Oct 2021.

[19] J. Herman and W. Usher. SALib: An open-source Python library for Sensitivity Analysis. *The Journal of Open Source Software*, 2(9), jan 2017.

[20] F. Hutter, H. H. Hoos, and K. Leyton-Brown. Sequential Model-Based Optimization for General Algorithm Configuration. In *International Conference on Learning and Intelligent Optimization*, pages 507–523. Springer, 2011.

[21] 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.

[22] 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.

[23] S. P. Kolodziej, M. Aznaveh, M. Bullock, J. David, T. A. Davis, M. Henderson, Y. Hu, and R. Sandstrom. The suitesparse matrix collection website interface. *Journal of Open Source Software*, 4(35):1244, 2019.

[24] 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.

[25] D. C. Liu and J. Nocedal. On the limited memory BFGS method for large scale optimization. *Mathematical Programming*, 45:503–528, 1989.

[26] Y. Liu, W. M. Sid-Lakhdar, O. Marques, X. Zhu, C. Meng, J. W. Demmel, and X. S. Li. GPTune: Multitask Learning for Autotuning Exascale Applications. In *Proceedings of the 26th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, PPoPP '21, page 234–246, New York, NY, USA, 2021. Association for Computing Machinery.

[27] H. Luo, Y. Cho, J. W. Demmel, X. S. Li, and Y. Liu. Hybrid Models for Mixed Variables in Bayesian Optimization. *arXiv preprint arXiv:2109.07563*, 2022.

[28] H. Luo, J. W. Demmel, Y. Cho, X. S. Li, and Y. Liu. Non-smooth Bayesian Optimization in Tuning Problems. *arXiv preprint arXiv:2109.07563*, 2021.

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

[30] NERSC. Science gateways. `https://www.nersc.gov/assets/Uploads/19-Science-Gateways.pdf`.

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

[32] H. Rakotoarison, M. Schoenauer, and M. Sebag. Automated Machine Learning with Monte-Carlo Tree Search. *arXiv:1906.00170 [cs, stat]*, 2019.

[33] C. E. Rasmussen. Gaussian processes in machine learning. In *Summer school on machine learning*, pages 63–71. Springer, 2003.

[34] B. Ru, A. S. Alvi, V. Nguyen, M. A. Osborne, and S. J. Roberts. Bayesian Optimisation over Multiple Continuous and Categorical Inputs. *arXiv:1906.08878 [cs, stat]*, 2020.

[35] A. Saltelli. Making best use of model evaluations to compute sensitivity indices. *Computer Physics Communications*, 145(2):280–297, 2002.

[36] A. Saltelli, P. Annoni, I. Azzini, F. Campolongo, M. Ratto, and S. Tarantola. Variance based sensitivity analysis of model output. design and estimator for the total sensitivity index. *Computer physics communications*, 181(2):259–270, 2010.

[37] 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.

[38] Slurm. Slurm workload manager. `https://slurm.schedmd.com/documentation.html`.

[39] I. M. Sobol. Global sensitivity indices for nonlinear mathematical models and their monte carlo estimates. *Mathematics and computers in simulation*, 55(1-3):271–280, 2001.

[40] P. Virtanen, R. Gommers, T. E. Oliphant, M. Haberland, T. Reddy, D. Cournapeau, E. Burovski, P. Peterson, W. Weckesser, J. Bright, S. J. van der Walt, M. Brett, J. Wilson, K. J. Millman, N. Mayorov, A. R. J. Nelson, E. Jones, R. Kern, E. Larson, C. J. Carey, İ. Polat, Y. Feng, E. W. Moore, J. VanderPlas, D. Laxalde, J. Perktold, R. Cimrman, I. Henriksen, E. A. Quintero, C. R. Harris, A. M. Archibald, A. H. Ribeiro, F. Pedregosa, P. van Mulbregt, and SciPy 1.0 Contributors. SciPy 1.0: Fundamental Algorithms for Scientific Computing in Python. *Nature Methods*, 17:261–272, 2020.

[41] X. Zhu, Y. Liu, P. Ghysels, D. Bindel, and X. S. Li. *GPTuneBand: Multi-task and Multi-fidelity Autotuning for Large-scale High Performance Computing Applications*, pages 1–13.