GPTune: Performance Autotuner for ECP Applications

Sherry Li, Yang Liu, Hengrui Luo
Lawrence Berkeley National Laboratory

James Demmel, Younghyun Cho
Univ. of California, Berkeley

April 14, 2021
Tutorial at ECP Annual Meeting

Acknowledgement

- Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

EXASCALE COMPUTING PROJECT

Team

Jim Demmel
UC Berkeley

Sherry Li
LBNL

Yang Liu
LBNL

Younghyun Cho
UC Berkeley

Hengrui Luo
LBNL

Past members:

Wissam Sid-Lakhdar
Univ. Tennessee

Osni Marques
LBNL

Mohsen Mahmouddi
Univ. Texas A&M

Chang Meng
Emory Univ.

Xinran Zhu
Cornell Univ.

https://github.com/gptune/GPTune

Plan

- Part I: Introduction of the tuning problems, methodology (20min)
  - Bayesian optimization framework, Gaussian process
  - GPTune software
- Part II: Demonstration of tuning the HPC application codes (20min)
- Part III: Recently developed features (30min)
  - History database
  - CK-GPTune
  - Clustered GP for non-smooth performance function surface
- Part IV: Hands-on experiments (20min)
  - Use Docker

Autotuning

- Problem

Given a target problem (task) and a parameterized code to solve it, find the parameter configuration (combination of parameter values) that optimizes (or improves) the code performance

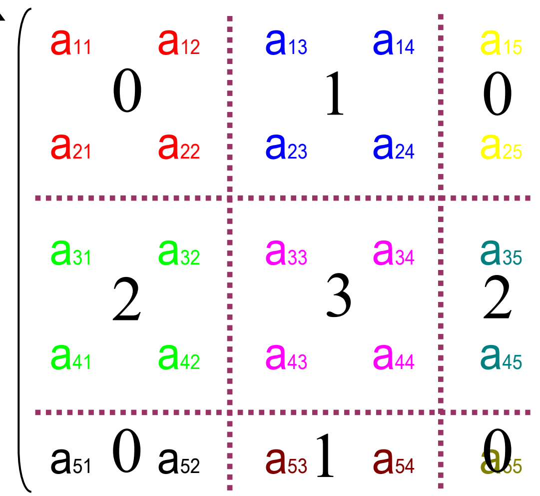
  - Metrics: solution time, memory or energy usage, etc. (or combined)

- ECP application codes are costly
  - Run on large supercomputers, for long periods of time

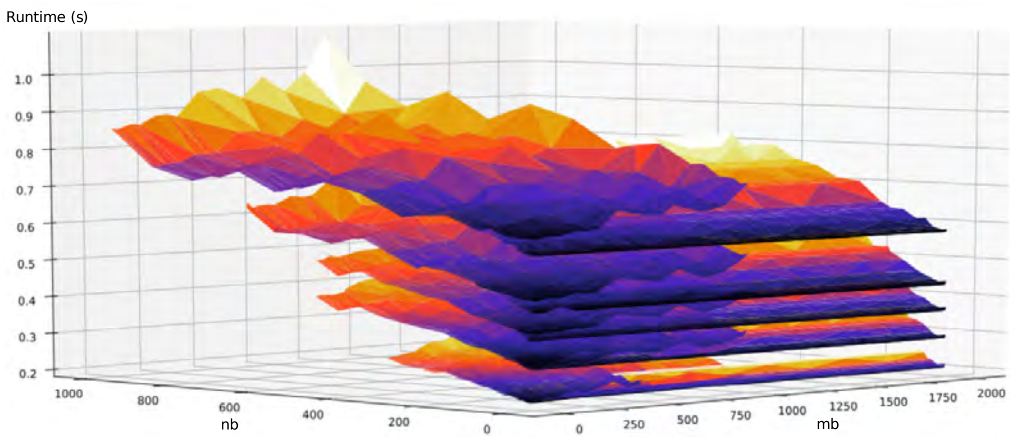
- Goal: make best use of the limited number of runs

Example: semi-exhaustive search

$m = n = 5, mb = nb = 2, p$=2



- Parallel dense QR factorization in ScaLAPACK
- 2D block-cyclic layout
- Tasks: {m, n}
- Parameters: {mb, nb, p} (nprocs=pxq)



1 node, 24 cores

$m = n = 2000$

$x-$axis: mb $y-$axis: nb

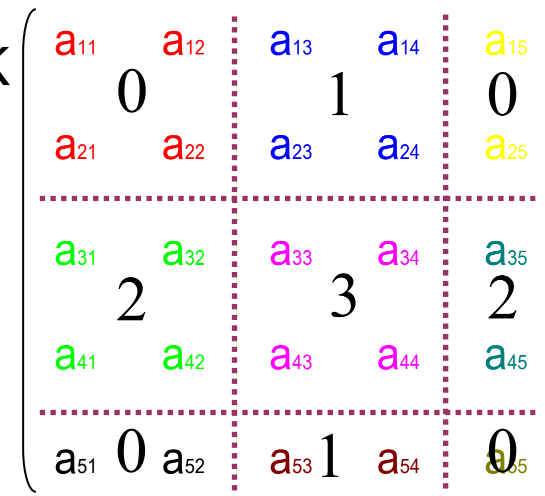
each layer is one pxq config.

- Rule of thumb for best performance (from algorithm viewpoint)
  - Process grid as square as possible
  - Blocks as square as possible

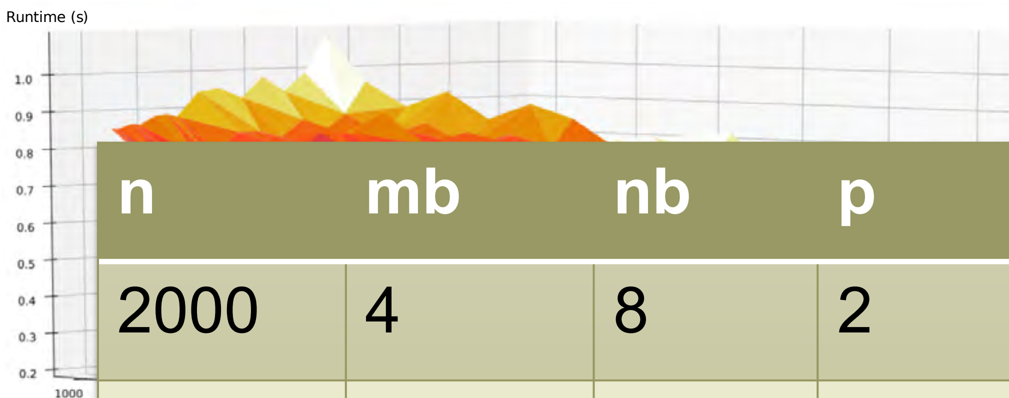
https://github.com/gptune/GPTune

Example: semi-exhaustive search

$m = n = 5, mb = nb = 2, p$=2



- Parallel dense QR factorization in ScaLAPACK
- 2D block-cyclic layout
- Tasks: {m, n}
- Parameters: {mb, nb, p} (nprocs=pxq)



1 node, 24 cores

n	mb	nb	p	q	time
2000	4	8	2	12	0.20
10000	1	26	16	192	1.61

nb

config.

https://github.com/gptune/GPTune

Characteristics of the optimization problems

- No analytical formulation of
  - objective function (runtime, memory, energy, …)
  - gradient
  - problem constraints

- Function evaluation == expensive application run (up to weeks!)
  - large variability related to hardware (e.g., network, disk I/O)

- Non-convex problems and non-linear constraints

- Discrete and continuous search spaces
  - Parameters can be Real, Integer, Categorical

Bayesian optimization

- Problem: $\underset{x}{\operatorname{argmin}}\, y(t, x)$, $\quad t$: task, x : parameter configuration
- Bayesian statistical inference is an iterative model-based approach
  - versatile framework for black-box derivative-free global optimization

Sampling: $\boldsymbol{y_1 \ldots y_{n_0}}$

Prior distribution

Surrogate Model $\boldsymbol{f(x)}$

$\min f \;\approx \min y$

Update Posterior Distribution

Search Next: $\boldsymbol{x_{n+1} \rightarrow y_{n+1}}$

Optimize Acquisition Function

$p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\mu_0, \Sigma_0)$

Multivariate normal dist.

likelihood

prior

Bayes Rule

$$p(y|x) = \frac{p(x|y)\, p(y)}{p(x)}$$

evidence

Update model with new data

https://github.com/gptune/GPTune

Bayesian optimization

- Problem: $\min_{x} y(t, x), \quad t \ : \text{task}, ; \quad x \ : \text{parameter configuration}$
- Bayesian statistical inference is an iterative model-based approach
  - versatile framework for black-box derivative-free global optimization

Sampling: $y_1 \ldots y_{n_0}$

Prior distribution

$p(\boldsymbol{y}|\boldsymbol{x}) = \mathcal{N}(\mu_0, \Sigma_0)$

Multivariate normal dist.

Surrogate Model $f(x)$

$\min f \ \approx \min y$

Update Posterior Distribution

Bayes Rule

$$p(y|x) = \frac{\overbrace{p(x|y)}^{\text{likelihood}}\ \overbrace{p(y)}^{\text{prior}}}{\underbrace{p(x)}_{\text{evidence}}}$$

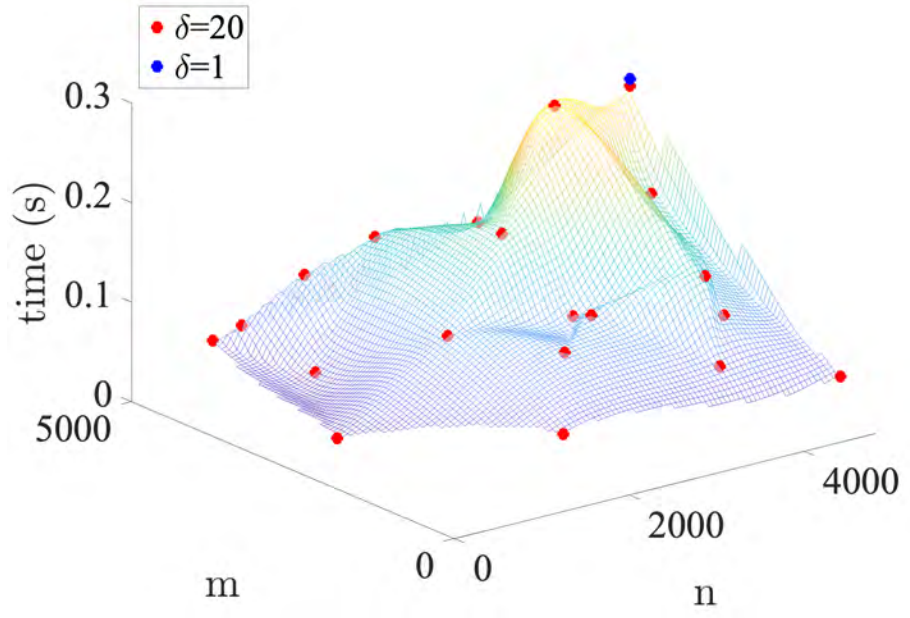
- GP regression
- LCM (multitask learning)

Search Next: $x_{n+1} \rightarrow y_{n+1}$

Optimize Acquisition Function

Update model with new data

- Expected Improvement (EI)
- Maximize EI: PSO, CMA-ES, …



Modeling

- Gaussian Process Regression

"Gaussian Processes for Machine Learning", Rasmussen and Williams 2006

- LCM: GP for vector-valued functions

"Kernels for Vector-Valued Functions", Alvarez, Rosasco, Lawrence, 2012

Gaussian Process

- GP defines a distribution over functions, and inference takes place in the space of functions
  - Every finite subset of variables follows multivariate normal distribution
- GP is specified by the mean function and covariance function $k(x, x')$ (kernel)

$$f(x) \sim GP\big(\mu(x), k(x, x')\big)$$
$$\mu(x) = \mathbb{E}[f(x]$$
$$k(x, x') = \mathbb{E}[(f(x) - \mu(x))(f(x') - \mu(x'))]$$

- Gaussian kernel (exponential square):

$$k(x, x') = \sigma^2 \exp(-\sum_{i=1}^{D} \frac{(x_i - x_i')^2}{l_i})$$

covariance is large if two points are close

(Can use other kernels)

GP model prediction

Given s observation pairs:

$$X = [x^1, x^2, \ldots, x^s] \quad Y = [y(x^1), \mathrm{y}(x^2), \ldots, y(x^s)]$$

Add new point x^*, posterior prob. distribution is : $p(y^*|X) = \mathcal{N}(\mu_*, \sigma_*^2)$

mean (prediction) and variance (confidence) for $\mathrm{y}(x^*)$ are:

$$\mu_* = \mu(X) + K(x^*, X)\, K(X, X)^{-1}\, (Y - \mu(X))$$
$$\sigma_*^2 = K(x^*, x^*) - K(x^*, X)\, K(X, X)^{-1} K(x^*, X)^T$$

Dimension of covariance matrix $K(X, X)$ = number of samples

Search Phase

- Where to place the new point(s)?
- Given a new sample point, need quickly update the model

Search for a point to maximize Acquisition Function
(… another optimization problem, but easier)

- Balance between exploitation and exploration
  - **Exploitation**: local search within promising regions
  - **Exploration**: global search of new regions with more uncertainty
- **Expected Improvement (EI)** – most commonly used AF.

  For a proposed point x_i^*, expected difference from current best is

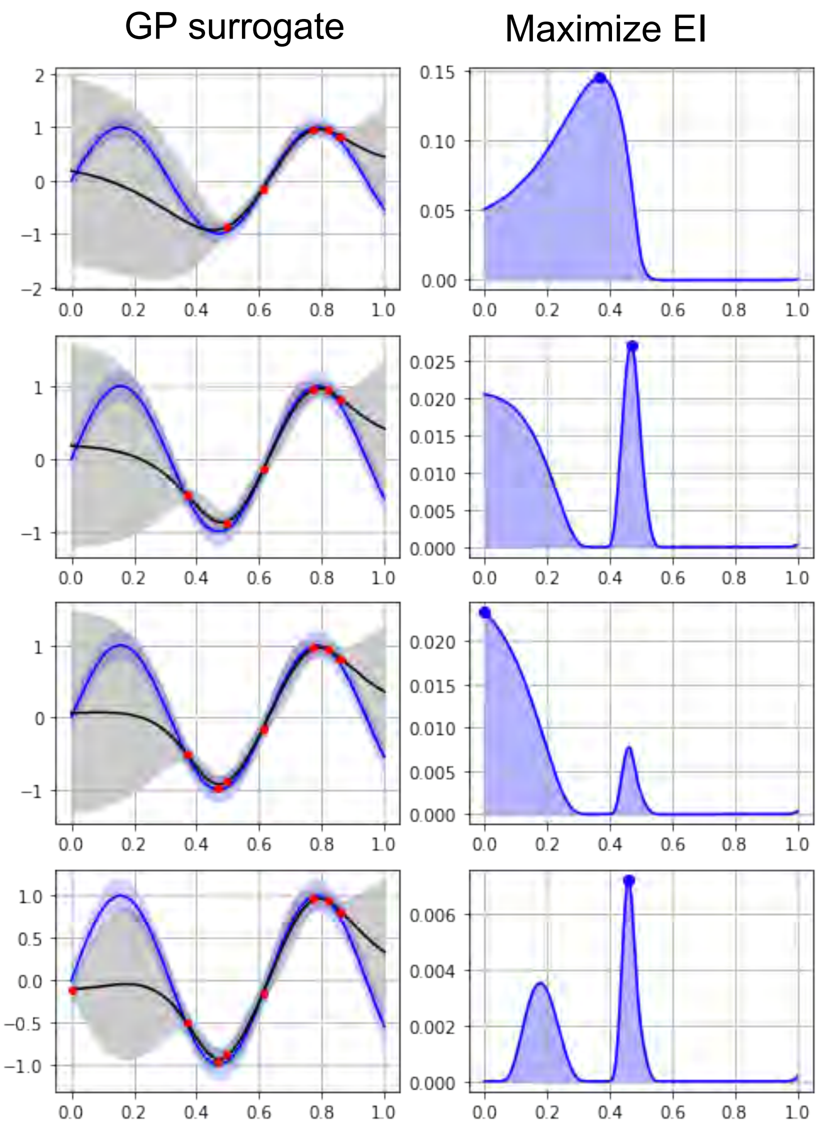
$$\Delta(x_i^*) = \mu_i^* - y_i^{min}$$

$$EI(x_i^*) = \mathbb{E}\left[\left[y_i^* - y_i^{min}\right]^+\right] = [\Delta(x_i^*)]^+ + \sigma_i^* \varphi\left(\frac{\Delta(x_i^*)}{\sigma_i^*}\right) - |\Delta(x_i^*)| \Phi\left(\frac{\Delta(x_i^*)}{\sigma_i^*}\right)$$

  - $\varphi(.)$: probability density function
  - $\Phi(.)$: cumulative distribution function

(Jones et al. 1998)

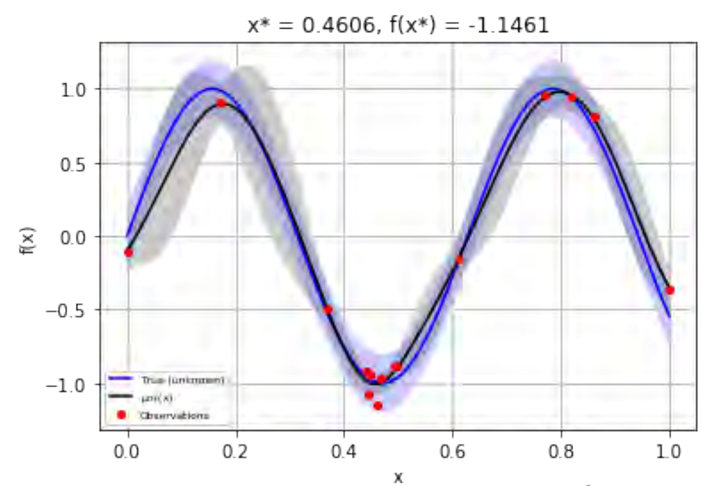
1D example: black-box function $y(x) = sin(10x)$



5 initial samples

4 additional steps

- Blue line: true function
- Red dots: function evaluations
- Black line: mean function of the fitted surrogate model
- Grey shaded area is 95% confidence interval



Multitask Learning Autotuning (MLA) – extending GP to vector-valued functions

"Kernels for Vector-Valued Functions", Alvarez, Rosasco, Lawrence, 2012

- Consider a set of correlated objective functions $\{y_i(X)\}_{i\in 1..\delta}$ (i.e., multiple tasks) and GP models $\{f_i(X)\}_{i\in 1..\delta}$
- Linear Coregionalization Model (LCM) attempts to build a joint model of the target functions through the underlying assumption of linear dependence on latent functions $\{u_q\}_{i\in 1..Q}$ (GP) encoding the shared behavior

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

with

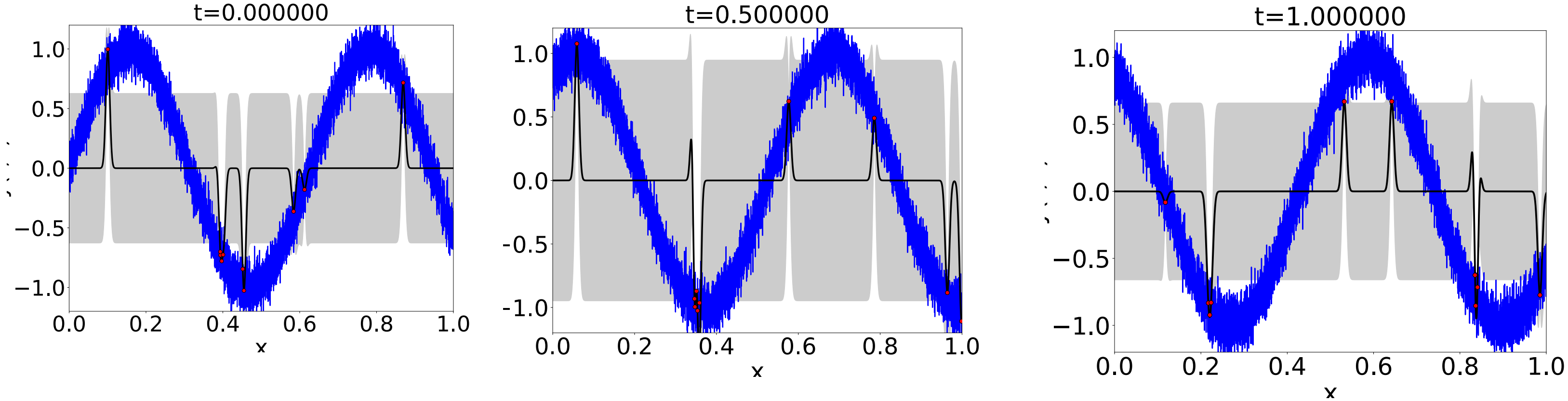
$$k_q(x, x') = \sigma_q^2 \exp\left(-\sum_{i=1}^{D} \frac{(x_i - x_i')^2}{l_i^q}\right)$$

"Big" covariance matrix size = number-of-tasks X number-of-samples-per-task

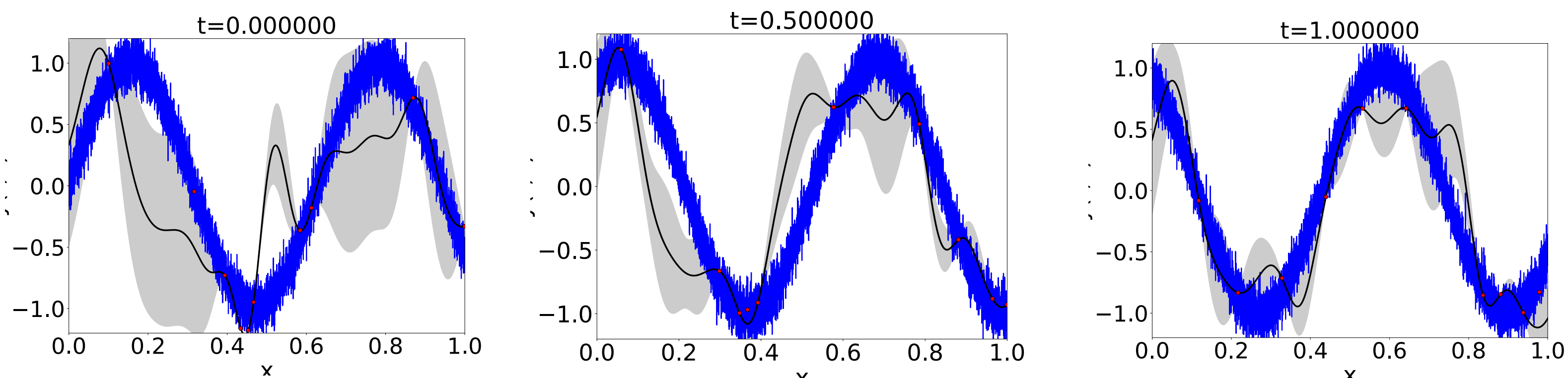
1D example with three correlated functions

$$y_{0.0}(x) = sin(10x) + \epsilon \,, \qquad \epsilon \sim \mathcal{N}(0, 0.1), \text{ Gaussian white noise}$$
$$y_{0.5}(x) = sin(10x + 1) + \epsilon$$
$$y_{1.0}(x) = sin(10x + 2) + \epsilon$$

Model separately



MLA



GPTune Software: github.com/gptune/GPTune ☺

- Python interface, leverage existing Python packages
  - GPy, scipy, scikit-learn, scikit-optimize, MPI4py, …
- C code for parallel matrix operations: BLAS, ScaLAPACK

- User input:
  - Task parameter input space ($\mathbb{IS}$): the space of tasks parameters
  - Tuning parameter space ($\mathbb{PS}$): the space of tuning parameters
    - (categorical, integer, real), and ranges
  - Output space ($\mathbb{OS}$): the space of objective function values
  - Define application as a black-box function
    - Python to C / Fortran interface

(Optionally)

- Define constraints in parameter search space
- Define performance models
- Choose a search method

Easy-to-Use Interface in Python

A model problem to illustrate user interface →

$$y(t, x) = sin(10x + 2t)$$

3 tasks: $t = 0, 0.5, 1.0$

Use Python classes to:

- Express arbitrary complex sets of constraints
- Provide arbitrary sets of tuning choices, as first class objects

```
from autotune.problem import *
from autotune.space import *
from autotune.search import *
Import numpy as np
```

Import internal Python classes

```
input_space = Space( [Real(0., 10., name="t")] )
parameter_space = Space( [Real(0., 1., name="x")] )
output_space = Space( [Real(-Inf, Inf, name="y")] )
```

Define application parameters (categorical, integer, real) and ranges

```
def objectives(point):
   t = point['t']
   x = point['x'];
   f = np.sin(10*x + 2*t)
   return [f*(1+np.random.uniform()*0.1]
```

Define application as a black-box function

```
constraints = {"cst1": "x >= 0. and x <= 1."}
```

Define constraints in parameter search space *[optional]*

```
def analytical_model1(point):
   f = np.sin(10*x + 2*t)
   return [f*(1+np.random.uniform()*0.1]
models = {'model1': analytical_model1}
```

Define performance models *[optional]*

```
problem = TuningProblem(input_space, parameter_space, output_space,
objectives , constraints, models)

Options['model_class'] = 'Model_LCM'
```

Choose a search method (GPTune by default)

```
gt = GPTune(problem, computer, data, options, … )
gt.MLA(NS, giventask, NI, NS1=int(NS/2))
```


https://github.com/gptune/GPTune

Advanced topics

- Part II: Tuning examples to show the following (20min)
  - Support Multitask Learning Autotuning (MLA)
  - Support multi-objective and multi-fidelity optimization
  - Support users' performance models to guide tuning process
  - Parallel performance on distributed-memory machines

- Part III: Recently developed features (30min)
  - History database
  - CK-GPTune
  - Clustered GP for non-smooth performance function surface

- Part IV: Hands-on experiments (20min)
  - Use Docker

APPENDIX

LCM model prediction – update posterior

Given δ tasks, each with s observations:

$$X = [\ \{x_1^1,\ x_1^2, \dots, x_1^s\}, \ . \ . \ . \ . \ , \{x_\delta^1,\ x_\delta^2, \dots, x_\delta^s\}\ ]$$

$$Y = [\{y_1(x_1^1),\ y_1(x_1^2), \dots, y_1(x_1^s)\}, \dots\ , \{y_1(x_1^1),\ y_1(x_1^2), \dots, y_1(x_1^s)\}\ ]$$

Add new point $X^* = [\ x_1^*,\ x_2^*, \dots, x_\delta^*]$

posterior prob. distribution is : $p(y^*|X) = \mathcal{N}(\mu_*, \sigma_*^2)$

mean (prediction) :

$$\mu_* = [\mu_1^*, \mu_2^*, \dots, \mu_\delta^*]^T \ = \ \mu(X) + K(X^*, X)\ K(X, X)^{-1}\ (Y - \mu(X))$$

variance (confidence) :

$$\sigma_*^2 = [\sigma_1^{*2}, \sigma_2^{*2}, \dots, \sigma_\delta^{*2}]^T = K(x^*, x^*) - K(x^*, X)\ K(X, X)^{-1} K(x^*, X)^T$$

"Big" covariance matrix includes both auto-covariance and cross-covariance

$$\Sigma\left(x_i^m, x_j^n\right) = \sum_{q=1}^{Q} a_{i,q} a_{j,q}\, k_q\left(x_i^m, x_j^n\right) + \ d_i \delta_{i,j}\, \delta_{m.n}$$

Learn hyper-parameters via gradient-based optimization

- Maximize marginal likelihood (== minimize log)

$$\log p(Y|X,\theta) = -\frac{1}{2}Y^T K^{-1} Y - \frac{1}{2}\log|K| - \frac{n}{2}\log(2\pi)$$

$$\theta \text{ is a collection of hyper}-\text{parameters } a_{i,q}, \dots$$

- Gradient-based optimization through L-BFGS-B
- Three levels of parallelism (MPI + OpenMP)
  - Spawn multiple MPI processes, each collaborating on separate instance of L-BFGS-B with a different random starting point of hyper-parameters. Only the best hyper-parameter among all processes are used
  - Each spawned process spawns more processes that collaborate in ScaLAPACK calls for K^{-1}
  - OpenMP threads collaborate on computation of and in multi-threaded BLAS

Search for a point to maximize Acquisition Function
(… another optimization problem, but easier)

- Balance between exploitation and exploration
  - **Exploitation**: local search within promising regions
  - **Exploration**: global search of new regions with more uncertainty
- **Expected Improvement (EI)** – most commonly used AF.

  For a proposed point x_i^*, expected difference from current best is

$$\Delta(x_i^*) = \mu_i^* - y_i^{min}$$

$$EI(x_i^*) = \mathbb{E}\left[\left[y_i^* - y_i^{min}\right]^+\right] = [\Delta(x_i^*)]^+ + \sigma_i^* \varphi\left(\frac{\Delta(x_i^*)}{\sigma_i^*}\right) - |\Delta(x_i^*)|\Phi\left(\frac{\Delta(x_i^*)}{\sigma_i^*}\right)$$

  - $\varphi(.)$: probability density function
  - $\Phi(.)$: cumulative distribution function

(Jones et al. 1998)

- We parallelize the search phase for every task (using MPI)
  - Find one (or several) new point(s) to evaluate by maximizing the EI, through a black-box optimization algorithm (PSO, CMA-ES, . . .)
  - Parallel optimization of EI (multi-threading through archipelago and island model in PAGMO)

GPTune Tutorial: Example Demonstration

*Yang Liu** [1], *Younghyun Cho* [2], *Hengrui Luo* [1,2], *Osni A. Marques* [1], *Xinran Zhu* [3], *Xiaoye S. Li* [1], *James Demmel* [2]

[1]Lawrence Berkeley National Laboratory

[2]University of California Berkeley

[3]Cornell University

Apr 14, 2021

ECP Annual Meeting

Space Definition and Data Implementation

Spaces

- Task input parameter space ($\mathbb{IS}$): the space of task parameters defining a problem
- Tuning parameter space ($\mathbb{PS}$): the space of tuning parameters
- Output space ($\mathbb{OS}$): the space of objective function values

Samples: data = Data(problem) with δ tasks and ϵ samples each

- data.I: length-δ list containing task samples
- data.P: length-δ list, element contains ϵ tuning parameter samples
- data.O: length-δ list, element contains ϵ objective evaluations
- data.D: length-δ list containing constants for data of each task

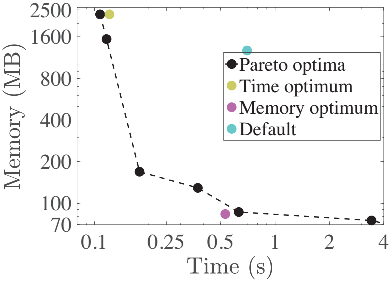
Outline

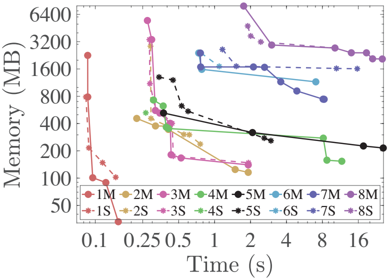
SuperLU_DIST: Multi-objective Tuning

- Experiment: ex= 'Fig.7_exp' in run_ppopp.sh.
- Plots: ex= 'Fig.7' in run_ppopp.sh.

Multi-objective EGO: One LCM per objective, NSGA-II in search

- $\mathbb{IS}$=[matrix name] $\mathbb{PS}$=[COLPERM, NSUP, NREL, nprows].
- Multi-objective: $\mathbb{OS} = [\mathit{time}, \mathit{memory}]$, single-objective: $\mathbb{OS} = [\mathit{time}]$ or $[\mathit{memory}]$. MLA $\delta = 8$ or single-task. 256 cores.





SuperLU_DIST: Multi-objective Tuning

Listing 1: pddrive_spawn.c

```c
int main(int argc, char *argv[]){
    float result[2]; // store factor time and memory
    MPI_Comm parent; MPI_Comm_get_parent(&parent);
    /* Read the input and parameters from command line arguments. */
    /* Perform computation */
    MPI_Reduce(result, MPI_BOTTOM, 1, MPI_FLOAT,MPI_MAX, 0, parent);
    MPI_Comm_disconnect(&parent);}
```

Listing 2: superlu_MLA_MO.py

```python
def objectives(point):
    # get input, tuning parameters and constants from point
    matrix = point['matrix']
    # pass some parameters through environment variables
    info = Info.Create()
    envstr= 'OMP_NUM_THREADS=%d\n' %(nthreads)
    info.Set('env',envstr)
    # spawn the executable with command line args and env
    comm = COMM_SELF.Spawn("./pddrive_spawn")
    # gather the return value using the inter-communicator
    ret = array('f', [0,0])
    comm.Reduce(sendbuf=None, recvbuf=ret,op=MAX,root=ROOT)
    comm.Disconnect()
    return ret
```

SuperLU_DIST: Multi-objective Tuning

Listing 3: superlu_MLA_MO.py (cont'd)

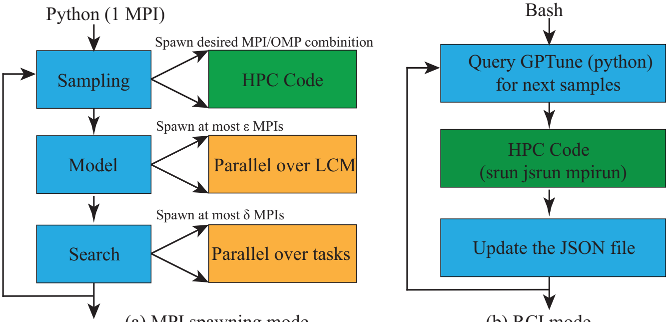
```
def cst1(NSUP,NREL):
    return NSUP >= NREL
def cst2(nprows,nodes,cores):
    return nodes * cores >= nprows
def main():
    # Get information in ./gptune/meta.json
    (machine,processor,nodes,cores)=GetMachineConfiguration()

    # Define spaces
    matrix=Categoricalnorm (["big","g4","g20"]) # Task
    COLPERM=Categoricalnorm (['2', '4']);NSUP=Integer(30, 300) # Tuning
    NREL=Integer(10, 40); nprows=Integer(1, nprocmax) # Tuning
    time=Real(-Inf , Inf); memory=Real(-Inf , Inf) # Objective
    IS=Space([matrix]);PS=Space([COLPERM,NSUP,NREL,nprows]);OS=Space([time,r

    # Define tuning problem
    constraints={"cst1":cst1,"cst2":cst2};constants={"c0":nodes}
    problem=TuningProblem(IS,PS,OS,objectives,constraints,constants)

    # Intialize the tuner
    data = Data(problem); gt = GPTune(problem, computer, data, options)

    # Perform MLA
    giventask = [["big"], ["g4"]]
    (data, model,stats) = gt.MLA(NS=NS, Igiven =giventask)
```

SuperLU_DIST: Reverse Communication Interface (RCI)

- Single-objective: uncomment line 370-376 in run_examples.sh.
- Multi-objective: uncomment line 378-384 in run_examples.sh.

RCI mode

- MPI-Spawn (OpenMPI) not required
- More flexible, portable (CrayMPICH, Spectrum MPI)



(a) MPI spawning mode (b) RCI mode

SuperLU_DIST: Reverse Communication Interface (RCI)

- Define GPTune meta data in Python with options['RCI_mode']=True
- Query GPTune, handle data with jq and invoke application in bash

Listing 4: superlu_MLA_MO_RCI.sh

```
obj1=time;obj2=memory # objectives defined in superlu_MLA_MO_RCI.py
db="gptune.db/SuperLU_DIST.json" # used to communicate with GPTune
more=1; while [ $more -eq 1 ];do # start the main loop
  # query GPTune for next sample points
  python ./superlu_MLA_MO_RCI.py -nrun $nrun
  idx=$( jq -r --arg v0 $obj1 '.func_eval|map(.evaluation_result[$v0]==null)'$db)
  if [ $idx = null ];then;more=0;fi
  while [ ! $idx = null ];do # loop over all samples requring eavluation
  ... # get the parameters into input_para and tuning_para
  matrix=${input_para[0]}  # get the task input parameters
  # get the tuning parameters
  COLPERM=${tuning_para[0]};NSUP=${tuning_para[1]}
  NREL=${tuning_para[2]};nprows=${tuning_para[3]}
  # call the application
  export OMP_NUM_THREADS=$cores;export NREL=$NREL
  export NSUP=$NSUP;nproc=$(($nodes*$cores))
  srun -n $nproc pddrive_spawn -r $nprows -p $COLPERM $matrix | tee log.out
  # get the result (for this example: search the runlog)
  result1=$(grep 'Factor time' log.out | grep -Eo '[+-]?[0-9]+([.][0-9]+)?')
  result2=$(grep 'Total MEM' log.out | grep -Eo '[+-]?[0-9]+([.][0-9]+)?')
  ... # write (result1,result2) back to the database file
  idx=$( jq -r --arg v0 $obj1 '.func_eval|map(.evaluation_result[$v0]==null)' $db);done
done
```

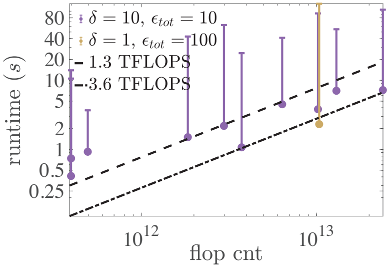
Outline

PDGEQRF: MLA v.s. Other Tuners

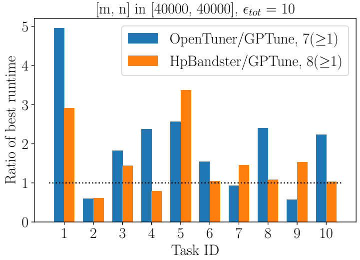
- Experiment: ex= 'Fig.6_exp' in run_ppopp.sh.
- Plots: ex= 'Fig.6' in run_ppopp.sh.

MLA vs. single-task vs. other tuners

- $\mathbb{IS}$=$[m, n]$, $\mathbb{PS}$=$[b_r, b_c, p, p_r]$, $\mathbb{OS}$=[runtime]
- L: Single-task: m:23324, n:26545, MLA: $m, n < 40000$. 2048 cores.
- R: $\delta = 10$ fixed matrices. 2048 cores.



(a) MLA vs Single-task



(b) GPTune vs Other tuners

PDGEQRF: MLA v.s. Other Tuners

- GPTune, opentuner, hpbandster: use common interface

Listing 5: scalapack_MLA.py

```
# Define objectives, constraints, spaces, options, computer
...
# Define the "autotune" interface to all tuners
problem = TuningProblem(IS, PS, OS, objectives, constraints)
# Call different tuners
if(TUNER_NAME=='GPTune'):
    gt = GPTune(problem, computer=computer, options=options)
    (data, model, stats) = gt.MLA(NS=NS, Igiven=giventask)
if(TUNER_NAME=='opentuner'):
    (data,stats)=OpenTuner(T=giventask,NS=NS,problem,computer)
if(TUNER_NAME=='hpbandster'):
    (data,stats)=HpBandSter(T=giventask,NS=NS,problem,computer)

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][argmin(data.O[tid])],'Oopt',min(data.O[tid])[0])
```

PDGEQRF: Incorporation of Coarse Performance Model

- Experiment: ex= 'Fig.4_exp' in run_ppopp.sh.
- Plots: ex= 'Fig.4' in run_ppopp.sh.

A coarse performance model $\tilde{y}(t,x)$ (per task) can be built into $\mathbb{PS}$: $x \rightarrow [x, \tilde{y}(t,x)]$. $\tilde{y}(t,x)$ can also be parameterized.

A simple performance model for PDGEQRF

$$\tilde{y}(t,x) = C_{flop} \times t_{flop} + C_{msg} \times t_{msg} + C_{vol} \times t_{vol} \tag{1}$$

with the number of floating point operations C_{flop}, the number of messages C_{msg} and the volume of messages C_{vol}

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

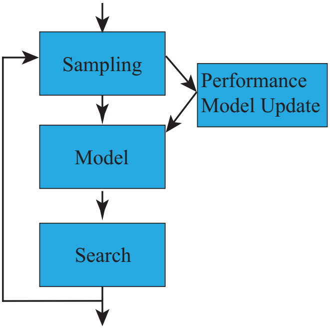
$$C_{msg} = 3n \log p_r + \frac{2n}{b_r} \log p_c \tag{3}$$

$$C_{vol} = \left(\frac{n^2}{p_c} + b_r n\right) \log p_r + \left(\frac{mn - n^2/2}{p_r} + \frac{b_r n}{2}\right) \log p_c \tag{4}$$

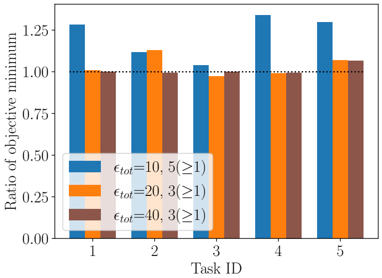
PDGEQRF: Incorporation of Coarse Performance Model

- Experiment: ex= 'Fig.4_exp' in run_ppopp.sh.
- Plots: ex= 'Fig.4' in run_ppopp.sh.

A coarse performance model $\tilde{y}(t, x)$ (per task) can be built into $\mathbb{PS}$: $x \to [x, \tilde{y}(t, x)]$. $\tilde{y}(t, x)$ can also be parameterized.



(a) Incorporate performance model



(b) PDGEQRF: ratio between best runtime with and without the performance model

PDGEQRF: Incorporation of Coarse Performance Model

Listing 6: scalapack_MLA_perfmodel.py

```
def models(point):
    ... # calculate Cflop,Cmsg,Cvol
    return [Cflop*(point['flop'])+Cmsg*(point['msg'])+Cvol*(point['vol'])]
def models_update(data):
    for i in range(len(data.I)):
    # update the hyperparameters of the performance model for each task
      X=np.array(data.P[i]);y=np.array(data.O[i])
      reg = LinearRegression(fit_intercept=False,normalize=False).fit(X, y)
      data.D[i]['flop']=reg.coef_[0][0]
      data.D[i]['msg']=reg.coef_[0][1]
      data.D[i]['vol']=reg.coef_[0][2]
def main():
    # Define objectives, constraints, spaces, options, computer
    ...
    # Define the "autotune" interface with a performance model 'models'
    problem = TuningProblem(IS, PS, OS, objectives, constraints, models)
    ntask=len(giventask)
    data = Data(problem,D=[{'flop': 0, 'msg': 0,'vol': 0}]*ntask)
    gt = GPTune(problem, computer, data, models_update)
    (data, model, stats) = gt.MLA(NS=NS, Igiven=giventask)
```

Outline

Hypre: GPTuneBand: Multi-fidelity Tuning

LCM + multi-armed bandit

Combine MLA with a multi-armed bandit strategy. Each arm has different starting fidelity and performs successive halving (SH). LCM can be built across arms and tasks.

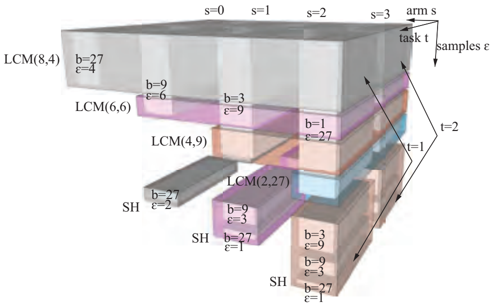


Figure: Illustration of multi-fidelity multi-task tuning with $\delta = 2$ tasks.

Hypre: GPTuneBand: Multi-fidelity Tuning

- Experiment: comment line 359-365 of run_examples.sh, set tuner=GPTuneBand, GPTune or hpbandster.

Multi-fidelity tuning of hypre

- Convection-diffusion equation on a n^3 grid:
$$-c\Delta u + a\nabla \cdot u = f$$
- $\mathbb{IS}$=[a, c], $\mathbb{PS}$=12 integer/real/categorical, $\mathbb{OS}$=[runtime]
- Fidelity/budget $\sim n^3$.

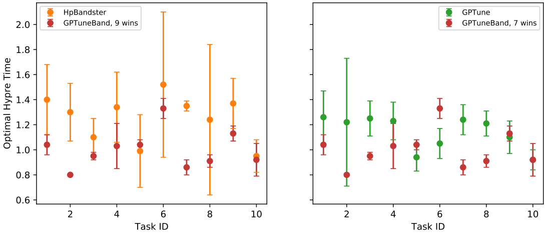


Figure: Comparison of GPTuneBand (multi-fidelity, MLA), GPTune (MLA), and HpBandster (multi-fidelity)

Hypre: GPTuneBand: Multi-fidelity Tuning

Listing 7: hypre_MB.py

```
def objectives(point):
    bmin = point['bmin'] # minimum budget (fidelity)
    bmax = point['bmax'] # maximum budget (fidelity)
    budget = point['budget'] # budget to be evaluated, asked by GPTune
    def budget_to_meshsize(b, nmin=10, nmax=100):
        k1 = (nmax**3-nmin**3)/(bmax-bmin)
        b1 = nmin**3 - k1
        return int((k1 * b + b1)**(1/3))

    n = budget_to_meshsize(budget)
    ... # Get task and tuning parameters
    ... # Call Hypre on a n^3 mesh and get the runtime
    return runtime

def main():
    ... # Define objectives, constraints, spaces, options, computer
    # Define the "autotune" interface with constants 'constants'
    constants={"bmin":bmin,"bmax":bmax}
    problem = TuningProblem(IS, PS, OS, objectives, constraints, constants)
    # Generate a list of tasks of interest
    giventask = ...
    # Call GPTuneBand
    gt = GPTune_MB(problem, computer, options)
    (data, stats, data_hist)=gt.MB_LCM(Igiven = giventask)
```

Outline

Other examples

- Use run_examples.sh and run_ppopp.sh (see hands-on instruction) to run more examples:
- GPTune-Demo
  - Parallel performance ./examples/GPTune-Demo/demo_parallelperformance.py
- Scalapack
  - PDGEQRF (RCI) ./examples/Scalapack-PDGEQRF_RCI/scalapack_MLA_RCI.sh
- STRUMPACK
  - 3D Poisson solver: ./examples/STRUMPACK/strumpack_MLA_Poisson3d.py
  - Kernel ridge regression: ./examples/STRUMPACK/strumpack_MLA_KRR.py
- MFEM
  - Maxwell: ./examples/MFEM/mfem_maxwell3d.py
  - Maxwell (RCI): ./examples/MFEM_RCI/mfem_maxwell3d_RCI.sh

Acknowledgments

This research was supported by the Exascale Computing Project (17-SC-20-SC), a joint project of the U.S. Department of Energy's Office of Science and National Nuclear Security Administration, responsible for delivering a capable exascale ecosystem, including software, applications, and hardware technology, to support the nation's exascale computing imperative.

SuperLU_DIST: GPU Tuning

- Experiment: TBD.

CUDA + MPI codes

- SuperLU_DIST: GPU factorization. $\mathbb{PS}$=[COLPERM, NSUP, NREL, N_GEMM, MAX_BUFFER, p_r]. Single-task: matrix_ACTIVSg70k_AC_00. 16 GPUs on 2 Cori nodes.

	COL	NSUP	NREL	N_GEMM	MAX_BUFFER	time (s)
Default	4	128	20	8192	500000	6.75
40 samples	2	771	107	65536	8388608	3.04

Table: Default and optimal tuning parameters using a single GPU.

	COL	NSUP	NREL	N_GEMM	MAX_BUFFER	p_r	time (s)
Default	4	128	20	8192	500000	4	5.61
40 samples	2	755	103	1048576	262144	1	2.64

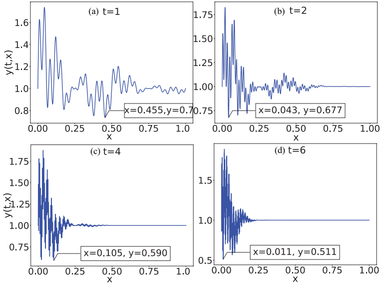
Table: Default and optimal tuning parameters using 16 GPUs.

Parallel Performance

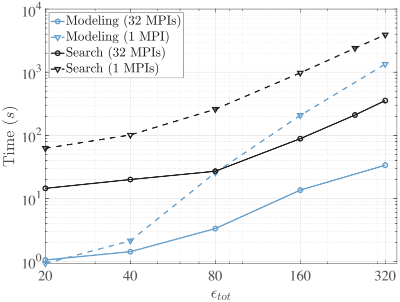
- Experiment: ex= 'Fig.3_exp' in run_ppopp.sh.
- Plots: ex= 'Fig.3' in run_ppopp.sh.

Consider an analytical function, t, x: task and tuning parameters, $\delta = 20$ tasks, ϵ_{tot}: number of samples per task.

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



Objective functions



Parallel performance

GPTune Tutorial: Recently Updated Features

Younghyun Cho, Hengrui Luo, Yang Liu,
Xiaoye S. Li, James Demmel

https://gptune.lbl.gov

Recently Updated Features

- **History database**
  - Allow GPTune to save/load historical performance data
  - Share your performance data with other users

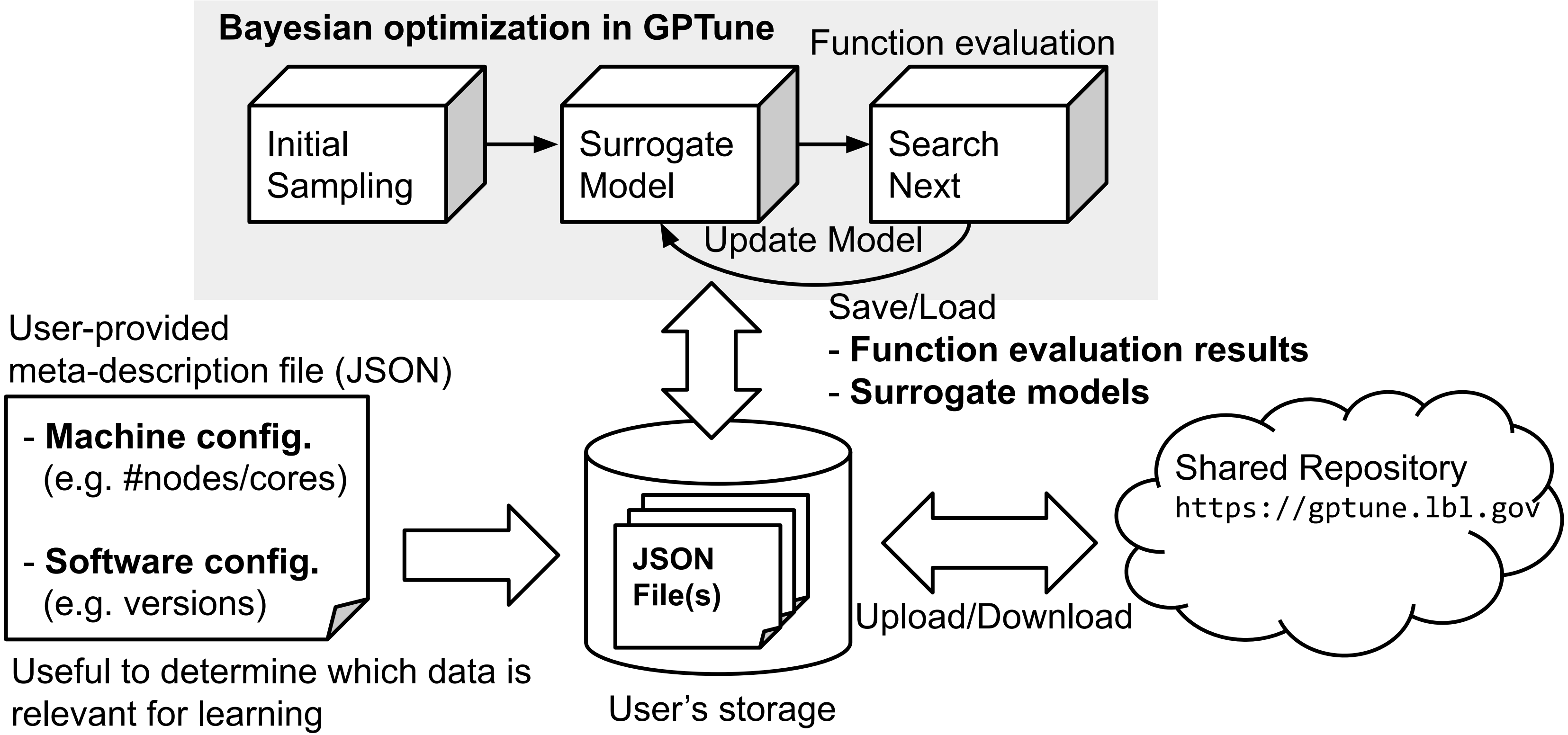
- **CK-GPTune**
  - Interface to run reproducible workflow from CK* with history database

- **Clustered GP**
  - Surrogate modeling for discontinuous performance function surface

* CK: Collective Knowledge (https://cknowledge.org)

History Database

- Allow GPTune to store/load historical performance data



Meta Description

- **History DB runs automatically with a meta-description file via common GPTune interfaces:**
  - MPI-spawning interface
  - Reverse communication interface

```
{
  "tuning_problem_name": "PDGEQRF",

  "machine_configuration": {
    "machine_name": "Cori",
    "Haswell": { "nodes": 1, "cores": 32 }
  }

  "software_configuration": {
    "scalapack": { "version_split": [2,1,0] }
  }
}
```

- Path to the meta description file: `$APP/.gptune/meta.json`

- Historical data is stored in `$APP/.gptune.db/`

* $APP: Path to the directory of the specific application (tuning problem)

Historical Data

Example: .gptune.db/PDGEQRF.json

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

each function evaluation result

each surrogate model

Given by meta file

Example function evaluation result

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

Example surrogate model

```
{
  "hyperparameters": [ 1.59484,
1295127.9634998, ... ],
  "model_stats": {
    "log_likelihood": -22.19576,
    "gradients": [ -9.37384, -7.43426,
... ],
    "iteration": 77
  },
}
```

- Unique ID and data creation time are automatically added for each data

History Database Use Cases

- **Checkpointing and restarting**
  - Useful for long autotuning processes, possible machine failures, limited job allocation times, etc.

- **Re-using pre-trained tuning results and surrogate models**
  - Better prediction with historical function evaluation results
  - In-depth analysis with the performance model

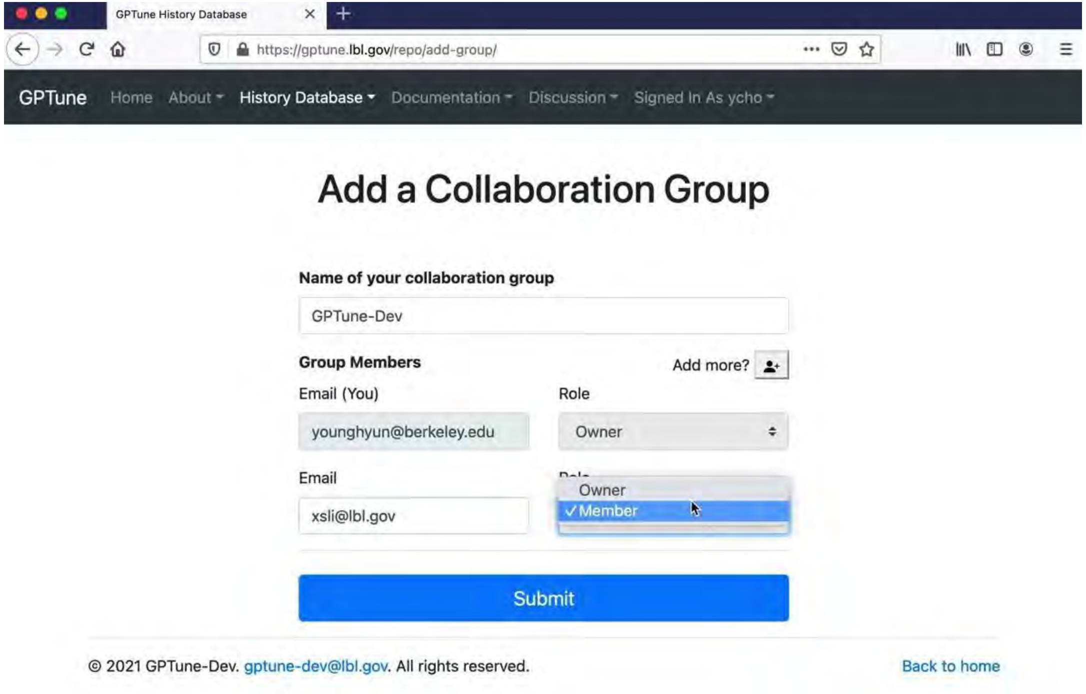
- **Crowd-tuning using our shared repository**
  - For popular applications, data can be collected over time by many users

Shared Repository

- A web interface using NERSC's resources (https://gptune.lbl.gov)

- Sign-up for free

- Level of accessibility
  - Publicly available
  - Registered users
  - Private
  - Sharing with specific users/groups



* click the image to play the recorded video

Downloading Data

1. Select the tuning application

2. Select machine/software configuration(s):
   The dashboard loads
   data obtained from
   the selected configurations.

3. Browse/Export results

* click the image to play the recorded video

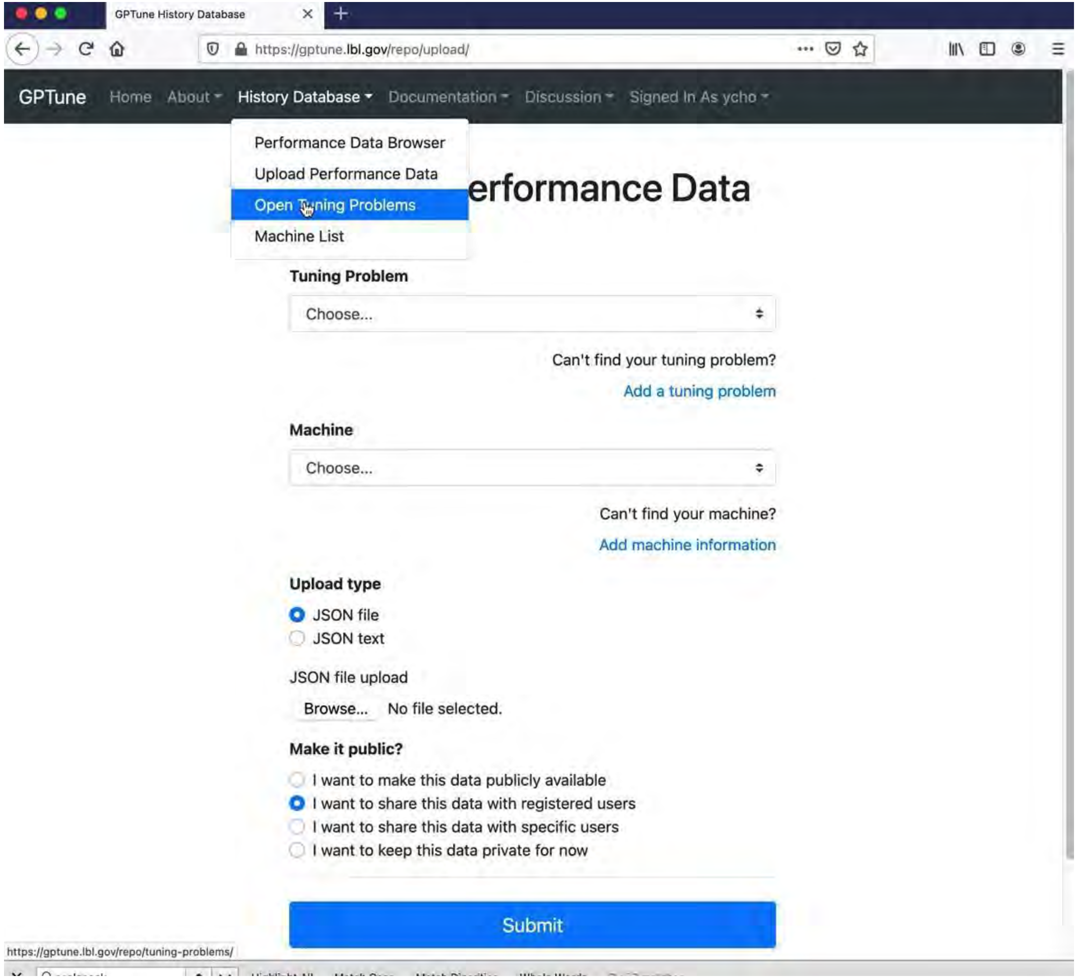
Uploading Data

1. Select your tuning problem and machine from the dropdown list

   If not exist: Add your tuning_problem/machine_info.

2. Select the accessibility

3. Upload JSON file/text



* click the image to play the recorded video

CK (Collective Knowledge) and CK-GPTune

- **CK (https://cknowledge.org) is an interface and tool for reproducible and automated workflows, and developed by cTuning foundation.**
  - Users write meta-description of their workflow using CK's syntax
  - CLI for auto-installation/compilation and running workflow
  - Software dependencies and versions are detected automatically
  - Can share the automated workflow with other user
- **CK-GPTune (https://github.com/gptune/CK-GPTune)**
  - CLI to run CK-enabled autotuning workflows with history database while taking the advantage of CK's software detection technology
  - Provide some examples of CK-enabled autotuning workflows e.g. PDQEQRF in ScaLAPACK, Pddrive routine in SuperLU_DIST

CLI: command line interface

Meta Description in CK

- **CK's meta description syntax**
  - How to compile/install? `compile_cmds`
  - How to run the workflow? `run_cmds`
  - Which software is required to compile/run? `compile_deps`, `run_deps`
- **Meta description for CK-GPTune**
  - No need to write software configuration. The information is detected automatically.
  - Need to provide `machine_configuration`

```
"machine_configuration": {
  "machine_name": "Cori",
  "Haswell": { "nodes": 1, "cores": 32 }
}
```

Append

Meta-description for reproducible workflow

```
{
  "compile_cmds": {
    "default": "./compile.sh"
  },
  "run_cmds": {
    "default": "./run.sh",
  },
  "compile_deps": {
    "compiler": {
      "name": "C Compiler",
      "tags": "compiler,lang-c"
    }
  },
  "run_deps": {
    "openmpi": {
      "name": "OpenMPI library",
      "tags":
      "version_from": [4,0,0]
    }
  }
}
```

How to Use CK-GPTune

- Installation & Setup (Python 3+ is required)
  1. `$ pip install ck --user`
  2. `$ ck pull repo --url=https://github.com/gptune/CK-GPTune`
  3. `$ ck detect soft:lib.gptune`

- If you have a CK-enabled GPTune autotuning workflow:
  - `$ ck MLA gptune --target=your_ck_program_name`

- If you want to run an example in CK-GPTune
  - `$ ck MLA gptune --target=`*`gptune-demo`*
  - https://github.com/gptune/CK-GPTune/tree/master/program

cGP: clustered Gaussian Process

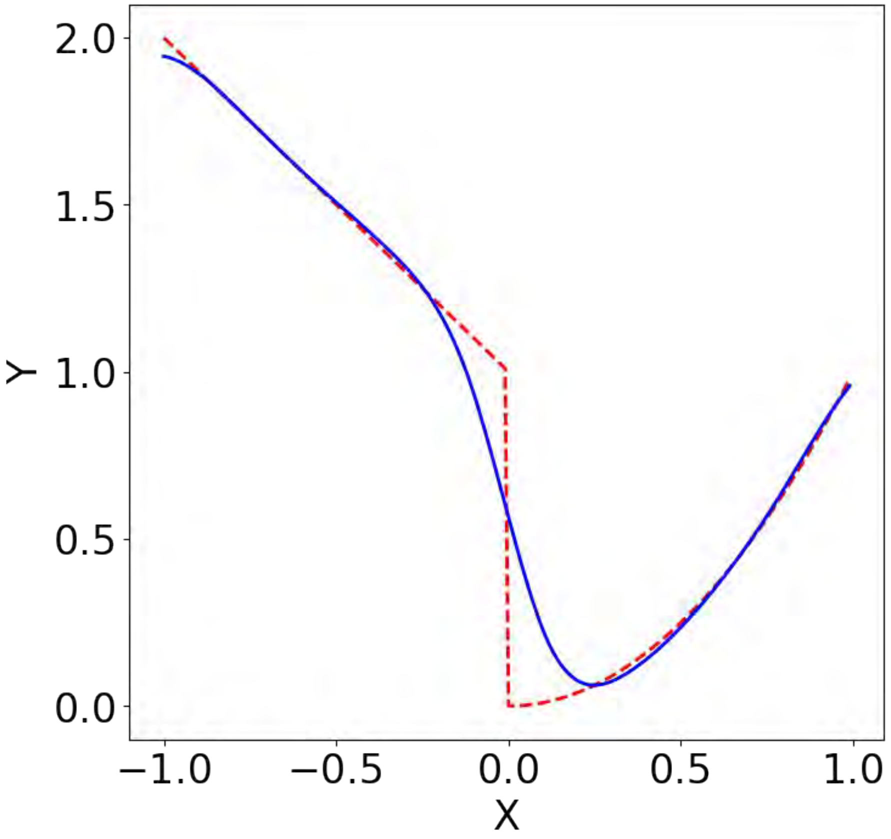
A smooth GP surrogate miss the minima x=0 of the non-smooth function *y=f(x)*:

$$f\left(x\right)=\begin{cases}-x+1 & x<0\\ x^2 & x\geq 0\end{cases}$$

world applications: matrix blocking, change-point, etc.

- Multiple strategies exist, but we address this within surrogate framework

  *Red line is the objective function f.
  *Blue line is a GP mean function.

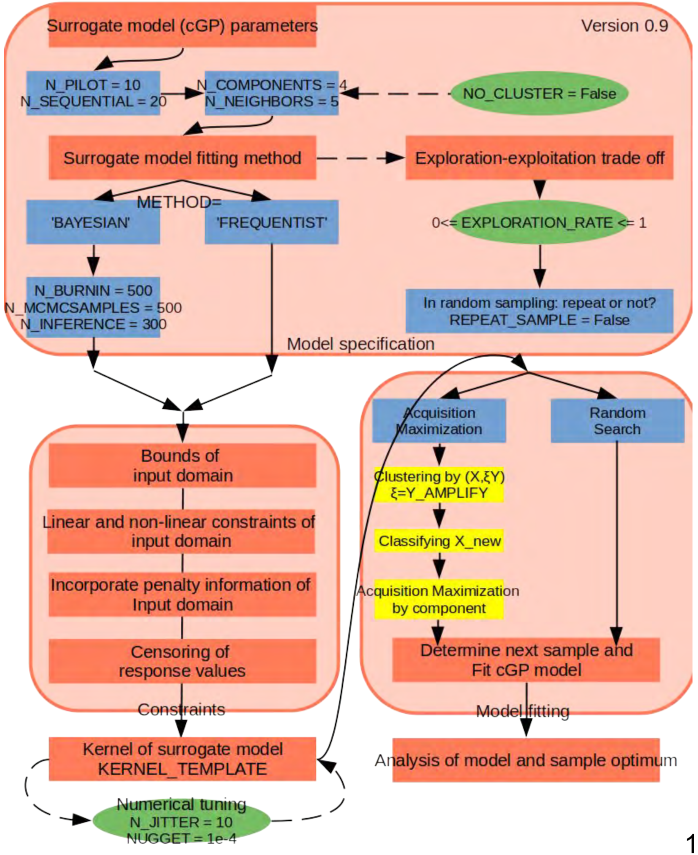


cGP: clustered Gaussian Process

- **Gaussian processes are smooth.**
  GP surrogate takes care of smooth black-box objective functions well and gives good optima.
- **What if my objective black-box function is rough?**
  - Change of regime
  - Jumps or drops in the tuning problem
  - Discontinuity in general functions
- **clustered Gaussian process are non-smooth.**
  - We use a GP surrogate model based on a partitioning of the variable domain.
  - The partitioning is induced by user-specified clustering and classification algorithms.
  - cGP surrogate takes care of non-smooth black-box objective functions.

cGP: flowchart and implementation

- **Key parameters of cGP**
  - maximal number of clusters. (N_COMPONENT) When this is 1, we are back to GP surrogate.
  - neighbors for classification. (N_NEIGHBORS) When this is 1, we use nearest neighbor partition.
  - exploration rate. (EXPLORATION_RATE) When this is 1, we always use acquisition maximization; when this is 0, we always use random search for next sample.



cGP: flowchart and implementation

- **Model**
  cGP can be used alone as a surrogate model
  - Minimum requirement built on python
  - Compatible with sklearn/scipy format
- **Computation**
  cGP also has natural computational advantage when handling large datum
  - Instead of fitting one large GP, cGP fits multiple smaller GPs automatically
  - Capture local behaviors in different components.
- **Incorporation**
  cGP would be incorporated into GPTune and handles non-smooth objective functions

Instruction for Hands-on Exercises

1. Install and start docker program on your laptop
    a. Linux
        i. Install
            - sudo apt install docker (Ubuntu), see https://docs.docker.com/engine/install/ubuntu/
            - sudo dnf install docker (Fedora), see https://docs.docker.com/engine/install/fedora/

        ii. Start Docker daemon
            - sudo systemctl start docker
            - sudo systemctl enable docker (Docker always starts at reboot)
    b. Windows
        - Sign up Docker Hub and download installer from https://docs.docker.com/docker-for-windows/install/
        - Launch Docker Desktop
        - Open powershell or cmd
    c. Mac
        - Sign up Docker Hub and download installer from https://docs.docker.com/docker-for-mac/install/
        - Launch Docker Desktop

2. Get the Docker image
    a. docker pull liuyangzhuan/gptune:2.4
    b. docker images
    (shows you what docker images you have available)

3. Run the Docker image (run as root)
    a. docker run -it liuyangzhuan/gptune:2.4
    (create a container from the image and run it in interactive mode)

4. Testing
At the root, edit run_examples.sh. At line 38, change “nodes=1” to the number of nodes on your machine (nodes=1 for most laptops/PCs). At line 247, change “cores=4” to the number of cores per node on your machine. Then keep a copy by “cp run_examples.sh run_examples.sh_backup”
    a. **GPTune-Demo**: cp run_examples.sh_backup run_examples.sh. Uncomment lines 291-297 and run the script. This example minimizes an

analytic function with one task input parameter *t* and tuning parameters *x*. The default setting generates 20 samples and 1 task. You can add command line options -nrun xxx and -ntask xxx after “python ./demo.py” to vary these numbers. The optimal tuning parameter and function value are printed after “Popt” and “Oopt”. The tuner runtime profile is printed after “stats:”. All function evaluation data are stored in ./examples/GPTune-Demo/gptune.db/GPTune-Demo.json.

b. **Scalapack-PDGEQRF**: cp run_examples.sh_backup run_examples.sh. Uncomment lines 298-303 and run the script. This example minimizes runtime of QR factorization of ntask=2 randomly generated matrices with sizes at most mmax=1000 x nmax=1000, and tuning parameters: blocking sizes, thread count, and MPI process grid. GPTune will generate run=40 samples per task. The tuner runtime profile is printed after “stats:”. All function evaluation data are stored in ./examples/Scalapack-PDGEQRF/gptune.db/PDGEQRF.json. The optimal tuning parameter and function value are printed after “Popt” and “Oopt” for each task “m:x n:y”. mmax, nmax, ntask, nrun can be changed at line 303.

c. **Scalapack-PDGEQRF_RCI**: cp run_examples.sh_backup run_examples.sh. Uncomment lines 359-364 and run the script. The application is the same as above. However, this example uses the reverse communication interface (RCI) of GPTune. nrun, mmax, nmax, ntask can be changed at line 364. All function evaluation data are stored in ./examples/Scalapack-PDGEQRF_RCI/gptune.db/PDGEQRF.json.

d. **More examples**: Setting “BuildExample=1” at line 7 and uncomment corresponding lines [SuperLU_DIST (line 307), STRUMPACK (line 315, 324), MFEM (line 333), ButterflyPACK (line 341)]. You can also work through all examples in run_ppopp.sh (see detailed comments there) to reproduce experiments and figures of the paper *GPTune: Multitask Learning for Autotuning Exascale Applications*, PPoPP21.

e. **Using shared repository**: Users can access our shared repository at https://gptune.lbl.gov and download/upload obtained function evaluation data. We encourage attendees to upload the generated GPTune-Demo’s JSON file (the demo example does not require specific software/machine configuration, so it is easy to try). We provide a tester account (ID: gptune-tester, Password: gptuneTester).
- After signing-in (https://gptune.lbl.gov/account/login/), access to the

upload form (https://gptune.lbl.gov/repo/upload/).
- Choose GPTune-Demo (defined by user "ycho") from the tuning problem list and "AnyMachine" from the machine list. Then, you can upload your GPTune-Demo JSON file.
- You will be able to view the submitted data by using our dashboard (https://gptune.lbl.gov/repo/dashboard/).

5. Other useful commands:
    a. Attach to a running container
       docker image ls (list all available images)
       docker container ls (list all running containers)
       docker attach container_id (attach to a running container with local changes
6. Installation without Docker. If you prefer not to use Docker, you can also install GPTune directly. The installation will take up to 2 hours depending on your system.
    a. Download GPTune:
        i. git clone https://github.com/gptune/GPTune.git
        ii. cd GPTune
    b. Install GPTune
        i. Ubuntu-like OS: bash config_cleanlinux.sh
        ii. Mac OS: Edit lines 6-9 of config_macbook.zsh to make sure they match the version numbers provided by homebrew. zsh config_macbook.zsh
        iii. NERSC Cori: bash config_cori.sh
    c. Run GPTune
        i. Edit top parts of run_examples.sh (and/or run_ppopp.sh) so that they match your system: Ubuntu (lines 34-38), Mac OS (lines 11-15), Cori (lines 18-22).
        ii. bash run_examples.sh (see section "4. Testing" above.)