Active-Learning-Based Surrogate Models for Empirical Performance Tuning

Prasanna Balaprakash
Mathematics and Computer Science Division
Argonne National Laboratory
pbalapra@mcs.anl.gov

Robert B. Gramacy
Booth School of Business
University of Chicago
rbgramacy@chicagobooth.edu

Stefan M. Wild
Mathematics and Computer Science Division
Argonne National Laboratory
wild@mcs.anl.gov

***Abstract*—Performance models have profound impact on hardware-software codesign, architectural explorations, and performance tuning of scientific applications. Developing algebraic performance models is becoming an increasingly challenging task. In such situations, a statistical surrogate-based performance model, fitted to a small number of input-output points obtained from empirical evaluation on the target machine, provides a range of benefits. Accurate surrogates can emulate the output of the expensive empirical evaluation at new inputs and therefore can be used to test and/or aid search, compiler, and autotuning algorithms. We present an iterative parallel algorithm that builds surrogate performance models for scientific kernels and workloads on single-core and multicore and multinode architectures. We tailor to our unique parallel environment an active learning heuristic popular in the literature on the sequential design of computer experiments in order to identify the code variants whose evaluations have the best potential to improve the surrogate. We use the proposed approach in a number of case studies to illustrate its effectiveness.**

I. Introduction

Automatic performance tuning is a response to the ever-growing complexity of high-performance computing (HPC) architectures and difficulties of manual tuning of scientific applications. *Autotuning* consists of identifying relevant code optimization techniques, assigning a range of parameter values using hardware expertise and application-specific knowledge, and then either exhaustively evaluating or searching this parameter space to find high-performing parameter configurations for the given machine.

The primary goal of performance models in autotuning has been to avoid running the corresponding code configuration on the target machine by predicting performance metrics of a given parameter configuration [1]. Such models are attractive, for example, for the design and development of search algorithms. Models can help prune large-scale search spaces [2], and their ease of evaluation allows for rigorous experimental comparisons among different search algorithms. Autotuning without models is costly because it requires running many different code variants on the target machine. Models also provide low setup cost for search algorithm developers who are typically not experts in autotuning. Moreover, accurate performance models allow researchers to perform large-scale experiments without requiring access to HPC systems except during the final phase of experimental analysis [1]. However, developing performance models is increasingly difficult because of the complexity of the HPC architectures and scientific codes [2], [3].

Analytical performance models, which use closed-form mathematical expressions for predicting performance metrics, have enjoyed significant success in the compiler optimization community for accelerating serial codes. However, this approach is limited by the quality and extrapolatory power of the mathematical model, which often fails to capture complex interactions between the code, runtime systems, and architecture. Moreover, developing a complex mathematical model requires a wide range of expertise in the target system architecture, programming models, and scientific applications. Consequently, analytical models are less well suited for highly specialized kernels and libraries for scientific applications that require portability, scalability, and performance [4], [5], [6].

When analytical performance models become too restrictive for a given scientific workload and HPC architecture, empirical performance modeling is an effective alternative [6], [7]. In this approach, a small subset of parameter configurations (code variants) is evaluated on the target machine to measure the required performance metrics, and a predictive model is built by using statistical approaches. We refer to these approximate models as *surrogate models*. Our focus in this paper is on empirical-based modeling, where we predict the outputs of new configurations on a single target machine, a task that we distinguish from performance modeling aimed at predicting outputs of a single configuration on new architectures.

Surrogate models for HPC workloads and kernels on CPU-based architectures have been based primarily on machine learning approaches [8], [9], [10]. Similar models for scientific kernels on multicore architectures have also been examined [3], [4], [11]. Artificial neural networks have been used to model power draw, execution time, and energy usage [5]. Boosted regression trees were adopted in [6] for obtaining online surrogate models for a GPU implementation of a spatial image filtering kernel. In all these works, the focus is the deployment of various algorithms for performance prediction. In a cluster computing environment, a naïve way to build any of these surrogate models is to sample a large number of configurations uniformly at random, evaluate these in parallel, and fit a model. Although such an asynchronous, embarrassingly parallel approach may seem like a holy grail in terms of work scalability, it can result in poor resource utilization due to poor model quality relative to the work required.

Faced with computationally expensive evaluations, a customary approach for developing surrogate models consists of sequential evaluation of parameter configurations. At each iteration, a model is fitted to all previously seen evaluations and is then used to decide which parameter configuration to evaluate next. In performance modeling, the sequential approach has been used to build surrogate models for resource allocation on networked utilities [12]. That work used a Plackett–Burman design [13], whose general applicability is limited because its linear-regression-based models are not appropriate for nonlinear response functions. Note that the sequential approach has been studied in the fields of statistics, applied mathematics, and machine learning, where it falls under the umbrella of design of experiments [14], simulation optimization [15], and active learning [16], respectively. Such an approach cannot take advantage of massively parallel environments.

The main difficulty of deploying active learning in parallel environments is the *a posteriori importance*: Given a model and a set of unevaluated parameter configurations, the active learning approach can query the model to select a subset (rather than a singleton) of the most-informative parameter configurations for parallel evaluation. However, as soon as a parameter configuration in the subset gets evaluated, the other configurations in the subset can become significantly less informative. Consequently, evaluating such configurations will likely not result in improving prediction accuracy and may therefore represent wasted evaluations.

In this paper we propose an iterative parallel algorithm that builds surrogate models for performance modeling. The novelty of the proposed approach consists of evaluating the parameter configurations in a sequence of batches to make use of multicore and multinode cluster computing environments in order to reduce the overall time required to obtain high-quality, fitted surrogate models. We address the a posteriori importance issue by tailoring the active learning to select a configuration based on prediction accuracy informed by other configurations to be evaluated in the same batch.

Active learning as a data acquisition scheme in surrogate modeling is still in its infancy. The closest related work [17] is from the design of experiments literature, where active learning is used to model the multiple outputs from a computational fluid dynamics simulator. This approach uses a nonlinear, treed Gaussian process (GP) modeling approach and takes into account asynchronous and batch mode evaluations. The approach we adopt uses dynamic regression trees, which have recently been shown to be more effective than treed GPs [18].

From the performance-modeling perspective, the main contributions of the paper are as follows. Previous work on surrogate models focused primarily on the adoption of machine learning approaches. To our knowledge, this is the first work on the design of a data acquisition strategy for building performance models with the objective of efficiently using HPC systems and minimizing the number of expensive evaluations on the target machine. Furthermore, most existing work in surrogate modeling deals with single-node architectures. In this paper, we show that our surrogate models can predict performances on massively parallel, leadership-class machines.

II. Dynamic trees

Dynamic trees [18] can be seen as regression trees combined with Bayesian inference. The former is a classical nonlinear regression approach that recursively partitions a multidimensional input space into a number of hyper-rectangles such that inputs with similar output values fall within the same hyper-rectangle. This partitioning scheme gives rise to a set of if-then-else rules that can be represented as a tree. Bayesian regression trees [19] are specified by a prior distribution on how the input space can be recursively partitioned and a likelihood comprising a product of simple, independent regression models applied in each partition. Together, these define a posterior distribution on the output space. Samples from the posterior distribution may then be obtained by simulation schemes such as Markov Chain Monte Carlo.

Dynamic trees specify a similar process for how trees evolve as new data arrive. At time t, after having seen data $(x, y)^t \equiv (x_1, y_1), \ldots, (x_t, y_t)$ and inferred a tree $\mathcal{T}_t|(x, y)^t$, a simple set of stochastic rules defines which $\mathcal{T}^{t+1}$ may be considered when (x_{t+1}, y_{t+1}) arrives. In this process, the new $\mathcal{T}_{t+1}$ must be identical to the old $\mathcal{T}_t$ except near the leaf node $\eta(x_{t+1})$ containing x_{t+1}. The process stochastically chooses from three local modifications based on support from y_{t+1} in the posterior distribution: *keep* $\eta(x_{t+1})$ unchanged in $\mathcal{T}_{t+1}$; *grow* a new split, making $\eta(x_{t+1})$ a parent of two new leaves in $\mathcal{T}_{t+1}$; or *prune* the tree to make the parent of $\eta(x_{t+1})$ a leaf in $\mathcal{T}_{t+1}$. A particle approach—essentially applying these rules independently to many similar trees grown stochastically on the same data—can reduce Monte Carlo error (via averaging) and lead to more accurate uncertainty quantification (by studying the spread of trees). Taking a sequential Monte Carlo, or “filtering,” approach that appropriately couples the particles/trees can offer further statistical efficiency gains.

Dynamic tree models have been shown to be competitive with several related nonparametric regression schemes in out-of-sample prediction exercises, both on batch data (with random data orderings) [18] and in online settings [20]. They have also proven useful for variable selection and input sensitivity analysis in contexts where the amount of data available traditionally swamps other comparators, such as GP models [21]. Software is available through the R package `dynaTree` [22].

The greatest potential of dynamic trees may lie in sequential design contexts, where the model fit is allowed to recommend the future inputs x_{t+1} (and corresponding outputs) on which it is trained. The original dynamic trees paper [18] suggested a heuristic called *active learning-Cohn* (ALC), which has been used to approximate maximum information designs in serial applications [23]. The ALC method involves choosing among new potential inputs x the one that gives the largest reduction in predictive variance averaged over the input space. For most response surface models (e.g., GPs), calculating this aggregated statistic requires numerical methods. However, conditional on the tree structure, it is analytic for dynamic trees—representing a large computational savings. The resulting designs allocate a heavier concentration of points in areas where the response surface is changing rapidly and put correspondingly fewer points in areas of the input space that are easier to predict.

Unfortunately, this methodology is tailored to one-at-a-time sequential design and model updating. That is, in a general iteration t, one recommends a new x_{t+1} based on an N-particle approximation $\{\mathcal{T}_t^{(i)}\}_{i=1}^N$; obtains y_{t+1}; and updates the particle approximation to $\{\mathcal{T}_{t+1}^{(i)}\}_{i=1}^N$. Many computer experiments are a hybrid of batch and sequential (see, e.g., [17]), which means that this scheme requires modification for the types of experiments we have in mind.

III. Parallel active learning with dynamic trees

The main obstacle that precludes direct adoption of any active learning scheme for parallel environments, whether based on dynamic trees or otherwise, is a posteriori importance. An active learning method, say ALC, can easily suggest a batch of parameter configurations for parallel evaluation. However, most appropriate spatial models for computer experiments (e.g., dynamic trees) facilitate the learning of a correlation structure between all outputs. Therefore, once any configuration in the batch finishes evaluation and the model fit is updated, uncertainty may greatly be reduced for the other points in the batch, possibly destroying their utility in improving the fit in a subsequent update (once their simulations have completed).

In order to address this issue, the active learning algorithm should first identify the single best candidate location x_i, say by ALC, and then identify other potential candidates whose predictive uncertainties are likely to be substantially reduced when x_i gets selected for the parallel evaluation. This latter set should be avoided when selecting the second candidate for the batch, and so on. One way this can be achieved is by treating the active learning model's prediction for x_i as "correct" as soon as x_i is selected for evaluation. We suggest using the fitted surrogate model to predict the output $\tilde{y}_i$ of x_i and update the model with $(x_i, \tilde{y}_i)$. This approach will reduce the predictive variance of configurations that depend on (x_i, y_i), which may be all configurations for stationary GPs or just those sharing a leaf with x_i in the `dynaTree` particle approximation. Since the predicted values from the fitted surrogate model may not be accurate—in particular during the initial iterations or when the algorithm moves to a previously unseen part of the input space—when the configurations in the batch are evaluated, the model has to unlearn the $(x_i, \tilde{y}_i)$ and relearn the value from the original evaluation (x_i, y_i).

Algorithm 1 (`ab-dynaTree`, where "a" and "b" stand for active learning and batch mode, respectively) summarizes the new iterative active learning algorithm. The symbols $\cup$, $-$, and $|\cdot|$ denote set union, difference, and cardinality operators, respectively. The algorithm takes a set $\mathcal{X}_p$ of unevaluated configurations and a maximum number of code-variant evaluations $n_{\max}$ as input. The two parameters of the algorithm are the subsample size n_s and batch size n_b, with $n_b \leq n_s$. The initialization phase (lines 1–4) consists of sampling n_s configurations at random from $\mathcal{X}_p$, obtaining a set of corresponding outputs, and using them as a training set to build a dynamic tree model $\mathcal{M}$. At each iteration, a configuration x_i that maximizes the ALC statistic is selected from the subsample set $\mathcal{X}_s$ and added to the batch set $\mathcal{X}_b$. The a posteriori importance issue within the same batch is addressed by using two models $\mathcal{M}$ and $\mathcal{M}_{\text{imp}}$. The output $\tilde{y}_i$ of x_i is predicted by using the imputed model $\mathcal{M}_{\text{imp}}$ and is updated with (x_i, $\tilde{y}_i$) (lines 11–12). As soon as the batch evaluation is over, the relearning phase is realized by copying the current active learning model $\mathcal{M}$ to $\mathcal{M}_{\text{imp}}$ (line 16). We can also use the training points ($\mathcal{X}_{\text{out}}$, $\mathcal{Y}_{\text{out}}$) obtained from active learning as a training set to build surrogate models using other regression approaches.

Algorithm 1 `ab-dynaTree`.

Input: pool of configurations $\mathcal{X}_p$, max evaluations $n_{\max}$, batch size $n_b \leq |\mathcal{X}_p|$, subsample size $n_s \geq n_b$

```
 1 $\mathcal{X}_{\text{out}} \leftarrow$ sample $\min\{n_s, n_{\max}\}$ distinct configurations from $\mathcal{X}_p$
 2 $\mathcal{Y}_{\text{out}} \leftarrow$ Evaluate_Parallel($\mathcal{X}_{\text{out}}$)
 3 $\mathcal{M} \leftarrow$ dynaTree ($\mathcal{X}_{\text{out}}$, $\mathcal{Y}_{\text{out}}$); $\mathcal{M}_{\text{imp}} \leftarrow \mathcal{M}$
 4 $\mathcal{X}_p \leftarrow \mathcal{X}_p - \mathcal{X}_{\text{out}}$; $\mathcal{X}_s \leftarrow \mathcal{X}_p$; $\mathcal{X}_b \leftarrow \emptyset$
 5 for $i \leftarrow n_s + 1$ to $n_{\max}$ do
     /* begin optional biased sampling */
 6     if $|\mathcal{X}_b| = 0$ then
 7         $\mathcal{Y}_p \leftarrow$ predict($\mathcal{M}$, $\mathcal{X}_p$)
 8         $\mathcal{X}_s \leftarrow$ biased_sampling($n_s$, $\mathcal{X}_p$, $\mathcal{Y}_p$)
 9     end if
     /* end optional biased sampling */
10     $x_i \leftarrow x \in \mathcal{X}_s$ that maximizes ALC statistic for $\mathcal{M}_{\text{imp}}$
11     $\tilde{y}_i \leftarrow$ predict($\mathcal{M}_{\text{imp}}$, $x_i$)
12     update $\mathcal{M}_{\text{imp}}$ with $(x_i, \tilde{y}_i)$
13     $\mathcal{X}_b \leftarrow \mathcal{X}_b \cup x_i$; $\mathcal{X}_s \leftarrow \mathcal{X}_s - x_i$
14     if $|\mathcal{X}_b| = n_b$ then
15         $\mathcal{Y}_b \leftarrow$ Evaluate_Parallel($\mathcal{X}_b$)
16         update $\mathcal{M}$ with $(\mathcal{X}_b, \mathcal{Y}_b)$; $\mathcal{M}_{\text{imp}} \leftarrow \mathcal{M}$
17         $\mathcal{X}_{\text{out}} \leftarrow \mathcal{X}_{\text{out}} \cup \mathcal{X}_b$; $\mathcal{Y}_{\text{out}} \leftarrow \mathcal{Y}_{\text{out}} \cup \mathcal{Y}_b$
18         $\mathcal{X}_p \leftarrow \mathcal{X}_p - \mathcal{X}_b$; $\mathcal{X}_b \leftarrow \emptyset$
19     end if
20 end for
```

Output: $\mathcal{X}_{\text{out}}$, $\mathcal{Y}_{\text{out}}$, $\mathcal{M}$

Before selecting the candidate configurations for a given batch, one can optionally select a subsample set according to a user-defined criterion (lines 6–9). Given a limited number of evaluations as a budget, and depending on the complexity of the unknown response function (for example, the number of disjoint local minima), there is a tradeoff between the number of training points and the prediction accuracy. In performance modeling for autotuning, one typically desires surrogates with higher prediction accuracy for configurations with good performance, rather than for configurations with poor performance. For this purpose, in the results described in the next section we take advantage of the optional procedure in Algorithm 1 to bias the sampling toward high-quality parameter configurations. At each iteration, instead of considering the entire pool of unevaluated configurations $\mathcal{X}_p$ for the ALC statistic computation, the algorithm first predicts the performance metric of each configuration in the pool, assigns a weight to each configuration that is inversely proportional to its performance metric, and selects a subset $\mathcal{X}_s$ by weighted sampling. The poor-quality configurations (e.g., those with longer run times or higher power consumption) are thus queried less frequently even if they are deemed to improve the overall prediction accuracy.

IV. Experimental Results

We now examine the effectiveness of `ab-dynaTree` in developing surrogate models of empirical performance data. Our goal is to determine whether active learning provides significant benefits over random search in parallel environments and whether the active learning approach is inherently tied to the `dynaTree` surrogate model.

To analyze the quality of evaluations performed by `ab-dynaTree`, we build models using three regression algorithms for the given problem. First we use the `dynaTree` algorithm (dT) with 10 repetitions, as recommended by the package authors, taking the prediction at each x as the mean of the 10 predictions. We compare this algorithm with two others: random forest (rf), a state-of-the-art and robust tree-based regression approach, and neural networks (nn), which has been shown to be effective for surrogate modeling [5]. For each algorithm, we consider two variants: *al*, in which the points obtained from `ab-dynaTree` are used for training the model, and *rs*, in which points are selected at random. The inclusion of rf and nn allows us to assess whether the obtained training points can also benefit other learning algorithms.

The parameter values of all these regression algorithms have a significant impact on the prediction accuracy. To avoid bias due to parameter tuning, we use the default parameter values (as in [22]) for dT and rf. Since our exploratory studies showed that the prediction accuracy of nn variants is poor, we use the tuned parameter values as suggested in [5]. We note that the results are biased toward nn variants because of this parameter tuning.

As a measure of prediction accuracy, we use root-mean-squared error (RMSE). We repeat each variant 10 times to reduce the impact of randomness throughout, and we consider the prediction accuracy of a variant as RMSE averaged over 10 repetitions. We also use a t-test to check whether the observed differences in the prediction accuracy of the variants are significant.

A. Modeling run times on serial codes

The first set of experiments was carried out on dedicated nodes of Fusion, a 320-node cluster at Argonne National Laboratory, comprising 2.6 GHz Intel Xeon processors with 36GB of RAM, under the stock Linux kernel v2.6.18.

We build surrogate models for problems from SPAPT [24], a collection of portable search problems in automatic performance tuning. Each problem in SPAPT is defined by a kernel, input size, set of tunable parameters, feasible set of possible parameter values, and default/initial configuration of these parameters. The kernels in SPAPT include elementary dense linear algebra, dense linear algebra solver, stencil code, and elementary statistical computing kernels.

The tuning parameter space includes loop unrolling ∈ [1,. . .,30], cache tiling ∈ [1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1024, 2048] (treated as [1,. . .,12]), and register tiling ∈ [1,. . .,32]. SPAPT problems also include binary parameters; however, here we set these to their nominal value (false) and consider only those parameters that can take several (integer) values, since these are primarily responsible for the size of search spaces in SPAPT. The resulting number of parameters ranges between 8 and 38, and the size of the search spaces range between 5.31×10^{10} and 1.24×10^{53}.

Given a problem, our goal is to build a surrogate model that can predict the mean run time of a parameter configuration x. The mean is computed over 35 generated code runs. For `ab-dynaTree`, we set the subsample size as $n_s = 100$ and batch size as $n_b = 50$. We generated 100,000 unevaluated configurations for $\mathcal{X}_p$ and set the maximum evaluations $n_{\max} = 5,000$.

TABLE I. RMSE AVERAGED OVER 10 REPLICATIONS ON THE $\mathcal{T}_{25\%}$ TEST SET FOR 2,500 (1,500 FOR BG/Q) TRAINING POINTS.

Problem	dT(al)	dT(rs)	nn(al)	nn(rs)	rf(al)	rf(rs)
A. SPAPT Run Times						
adi	0.021	*0.025*	*0.034*	*0.031*	*0.022*	*0.025*
atax	0.045	*0.057*	*0.064*	*0.072*	*0.056*	*0.069*
bicgkernel	0.021	*0.024*	*0.038*	*0.043*	*0.032*	*0.038*
correlation	0.060	*0.066*	*0.212*	*0.199*	**0.053**	**0.057**
covariance	0.055	*0.064*	*0.104*	*0.114*	*0.059*	*0.072*
dgemv3	0.057	*0.069*	*0.100*	*0.137*	*0.065*	*0.077*
gemver	0.100	*0.120*	*0.155*	*0.180*	*0.103*	*0.132*
hessian	0.045	*0.054*	*0.059*	*0.070*	*0.070*	*0.094*
jacobi	0.029	*0.045*	*0.058*	*0.057*	*0.044*	*0.053*
lu	0.037	*0.060*	*0.072*	*0.084*	*0.050*	*0.067*
mm	0.064	*0.079*	*0.078*	*0.079*	**0.061**	*0.075*
mvt	0.032	*0.036*	*0.044*	*0.053*	*0.044*	*0.053*
seidel	0.076	*0.097*	*0.092*	*0.098*	*0.080*	*0.095*
stencil3d	0.080	*0.100*	*0.100*	*0.122*	*0.084*	*0.105*
B. Power						
lu(cpu)	0.017	*0.022*	*0.119*	*0.116*	*0.018*	*0.029*
lu(dimm)	0.016	*0.021*	*0.083*	*0.096*	*0.034*	*0.055*
mm(cpu)	0.024	*0.031*	0.023	0.024	*0.036*	*0.045*
mm(dimm)	0.034	*0.051*	*0.074*	*0.095*	*0.057*	*0.076*
stencil(cpu)	0.015	*0.016*	**0.014**	**0.014**	*0.019*	*0.023*
stencil(dimm)	0.041	*0.048*	0.039	*0.050*	0.040	*0.052*
C. BG/Q						
cg(mflops)	0.000	*0.000*	*0.005*	*0.010*	*0.015*	*0.024*
cg(time)	0.000	0.001	*0.002*	*0.003*	*0.007*	*0.016*
dot(mflops)	0.003	0.003	*0.007*	*0.012*	*0.014*	*0.018*
dot(time)	0.001	*0.001*	*0.003*	*0.004*	*0.005*	*0.011*
matvec(mflops)	0.001	*0.001*	*0.004*	*0.006*	*0.013*	*0.024*
matvec(time)	0.001	0.001	*0.002*	*0.003*	*0.007*	*0.014*
waxpy(mflops)	0.001	*0.001*	*0.003*	*0.007*	*0.013*	*0.030*
waxpy(time)	0.000	0.001	*0.002*	*0.004*	*0.007*	*0.015*

Note: The value is typeset in *italics* (**bold**) when a variant is significantly worse (better) than dT(al) according to a t-test with significance (alpha) level 0.05.

To allow cross-comparison of prediction accuracy between the problems, we scale the run time values for each problem: each y_i is divided by $y_{i_{\max}}$, where $y_{i_{\max}}$ is the maximum run time from $\mathcal{Y}_{\text{out}}$. For the active learning variants, we consider the first 2,500 points from ($\mathcal{X}_{\text{out}}$,$\mathcal{Y}_{\text{out}}$) as the training set to build the surrogate model. We derive two test sets from the remaining 2,500 points: (i) the subset of points $\mathcal{T}_{25\%}$ from the training set whose mean run times are within the lower 25% quantile of the empirical distribution for the 5,000 run times in $\mathcal{Y}_{\text{out}}$; and (ii) a set $\mathcal{T}_{100\%}$ of 1,000 randomly generated points. For the random sampling variants, we use the same test sets $\mathcal{T}_{25\%}$ and $\mathcal{T}_{100\%}$ but a different training set, with the 2,500 points for training being randomly chosen from $(\mathcal{X}_{\text{out}}, \mathcal{Y}_{\text{out}}) - \mathcal{T}_{25\%}$. Since the training points of random sampling variants are not entirely random, their perceived effectiveness may be artificially higher than one could expect in reality.

Table I-A shows RMSEs averaged over 10 replications for 2,500 training points and tested on $\mathcal{T}_{25\%}$. The results show that, except for bicgkernel, dT(al) obtains lower average RMSE than does dT(rs). The trend is similar on the nn variants. We can also observe that the dT variants completely dominate the nn variants despite the latter using tuned parameter values. The key advantage of dT(al) comes from it requiring relatively fewer evaluations to achieve a smaller RMSE. This is illustrated in Fig. 1, which shows the RMSE as a function of the number of training points. Since the results of the nn variants are poor, we did not include them in the figure. The results

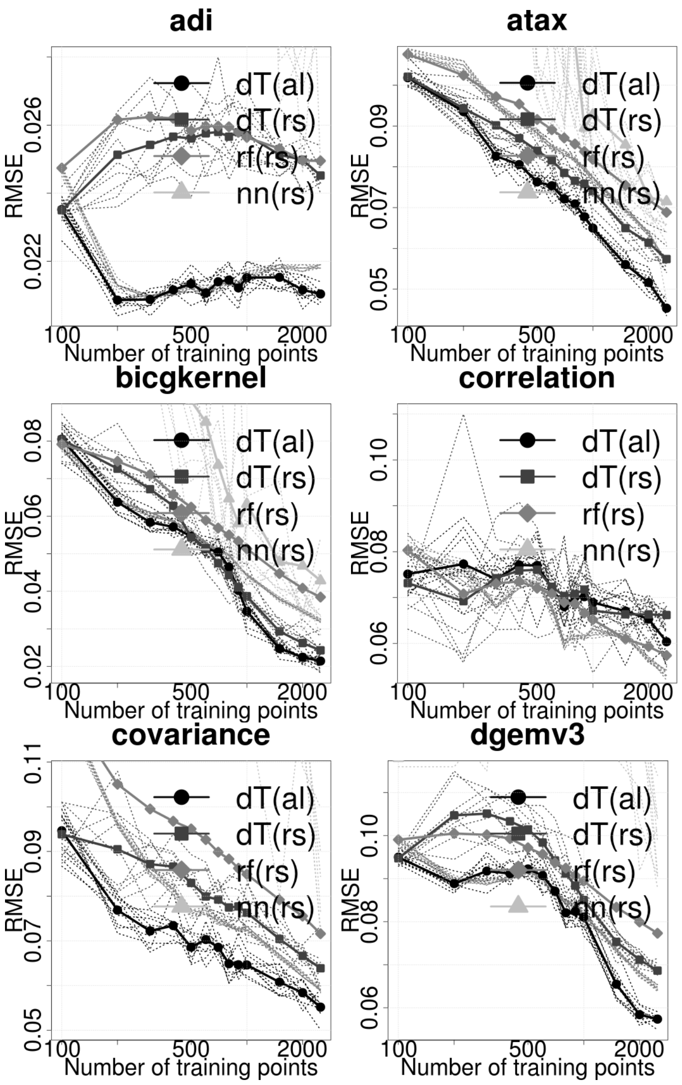

Fig. 1. RMSE for the first six SPAPT problems on the test set $\mathcal{T}_{25\%}$. The dotted lines represent the RMSE for each replication and the bold lines represent the mean RMSE over 10 replications.

show that dT(al) achieves a lower RMSE than does dT(al) with relatively fewer training points. In Fig. 2, we compare the number of evaluations required by the variants to reach the RMSE obtained by dT(al) (with 2,500 evaluations). On 5 out of 10 problems, dT(al) reaches the RMSE of dT(rs) within 1,000 training points. Only for bicgkernel and mvt is there no significant difference in the number of training points.

B. Modeling power consumption of serial code kernels

In this section, we focus on modeling the power consumption of serial code kernels. We obtained the component-level (CPU and DIMM) power consumption data used in [5] for mm, stencil, and lu computations. The data was collected on an Intel Xeon E5530 workstation with two quad-core processors, where each core has 32KB L1 cache and 256KB L2 cache; see [5] for further details.

The mm, stencil, and lu kernels have 7, 5, and 11 parameters and search space sizes of 6.5×10^5, 5.6×10^{10}, and 8.3×10^{10} configurations, respectively. The parameters are tiles, unroll, input size, and clock frequency. The data set comprises 8,285, 4,900, and 9,700 randomly sampled configurations for mm, stencil, and lu, respectively. We stored these configurations and their corresponding outputs in a lookup table and simulated the batch mode for `ab-dynaTree`, where the randomly sampled configurations were given as the configuration pool.

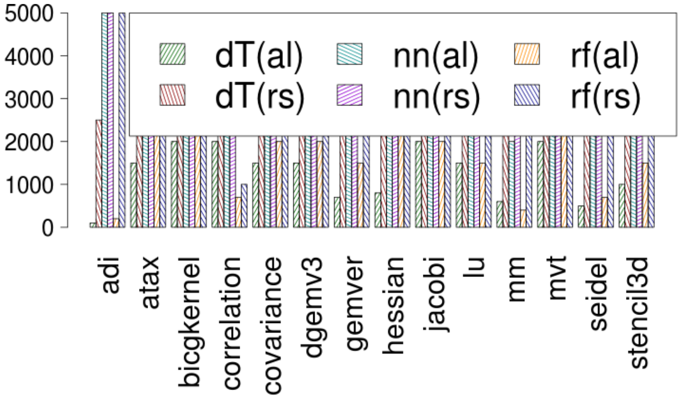

Fig. 2. Number of evaluations required to reach the RMSE of dT(rs) (with 2,500 evaluations) for SPAPT problems on the test set $\mathcal{T}_{25\%}$.

The results are shown in Fig. 3. We observe that al variants obtain lower RMSE when compared with rs variants, suggesting that the active learning approach is beneficial. The results from Table I-B show that dT(al) obtains significantly lower RMSE than do other variants except for stencil, where nn variants obtain slightly lower RMSE than does dT(al). The computational savings are shown in Fig. 4, where we can see that dT(al) requires fewer than 1,000 training points to reach the RMSE of dT(rs). The correlation between observed and predicted values of dT(al) is shown in Fig. 5.

C. Modeling run time and FLOPS of MPI code

In this section, we model run time and FLOPS of miniFE, a mini application that comprises the most significant performance characteristics of an implicit finite element method using a simple un-preconditioned conjugate-gradient (cg) solver; see [25] for details. We used the following parameters: number of nodes $\in \{8, 16, 32, 64, \ldots, 4096\}$, number of processes per node $\in \{2, 4, 8, 16, 32, 64\}$, % of artificial load imbalance $\in \{5, 10, 20, 30, \ldots, 90\}$, overlapping communication and computation indicator $\in \{0, 1\}$, and size of the box domain (10 values from 100 to 500 considered at equal intervals). The total number of configurations is 12,000. The experiments were carried out on Mira, a 10PF IBM Blue Gene/Q at the Argonne Leadership Computing Facility. It has 49,152 nodes organized in 48 cabinets, where each node comprises 16 cores of 1.6 GHz PowerPC A2 and 16GB of DDR3 memory.

We model run times and FLOPS of the cg kernel in miniFE. For each evaluation, in addition to the run time, we record the FLOPS taken by cg and the run time and FLOPS taken by the dot product (dot), matrix-vector product (matvec), and vector-scalar product (waxpy) kernels within cg. We run `ab-dynaTree` only once with the objective of building a model for the run time and with the maximum number of evaluations of 2,500. In this study, we use the same evaluations for modeling both the FLOPS and run time metrics.

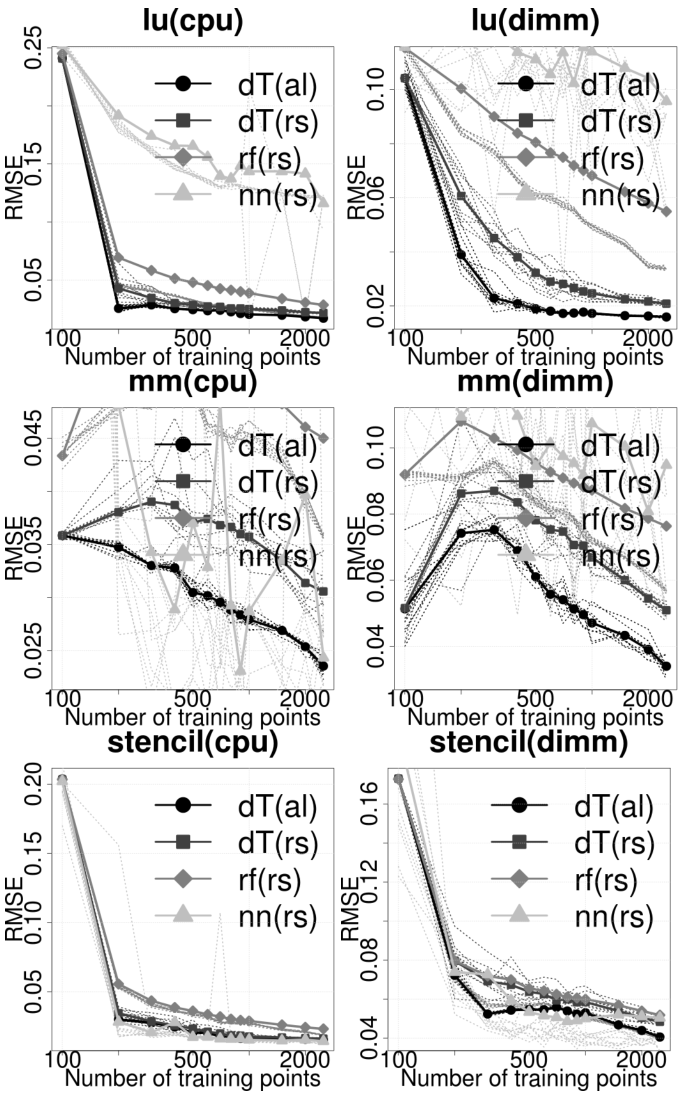

Fig. 3. RMSE for lu, mm, and stencil CPU and DIMM power on the test set $\mathcal{T}_{25\%}$. The dotted lines represent the RMSE for each replication and the bold lines represent the mean RMSE over 10 replications.

For the validation experiments, the al variants use the first 1,500 configurations obtained from ab-dynaTree. The results are shown in Fig. 6, Fig. 7, and Table I-D. The differences in mean RMSE between dT(al) and dT(rs) are small (in the range of 0.001 to 0.003) and the t-test shows that, out of the 8 problems, only on 4 problems does dT(al) obtain an RMSE significantly lower than that of dT(rs). On the remaining four problems, we do not have significant evidence to say that one is better than the other. However, the differences between the al and rs variants of nn and rf are larger and clearly show that adoption of ab-dynaTree results in reducing RMSE. The overall low RMSE values and illustrative scatter plots of cg and dot in Fig. 8 show that the training points, though obtained with the goal of modeling run times of cg, are highly effective for obtaining surrogate models for the other 7 metrics.

D. Experiments on batch size

In this section, we study the impact of batch size n_b in ab-dynaTree on the RMSE. We used the same experimental setting and data as described IV-B. For the ab-dynaTree, we consider values in $\{1, 50, 100, 200\}$. Note that $n_b = 1$ corresponds to the sequential version. Thus, each iteration of ab-dynaTree consists in n_b parallel evaluations. The subsample size is set to $2 \times n_b$ except for the sequential version in which it is set to 100. The training points obtained from ab-dynaTree is given to dT(al) and all these versions start with the initial sample size of 100. We assume each evaluation takes the same time unit. The results are given in Figure 9.

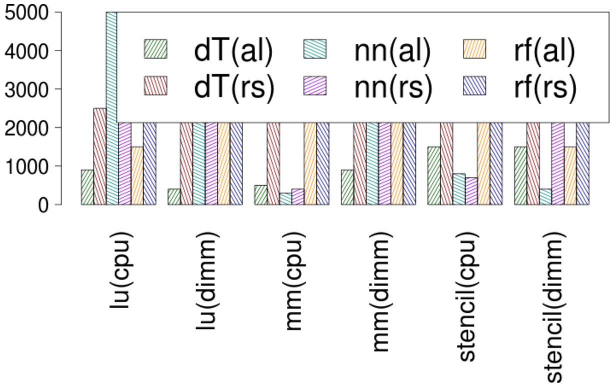

Fig. 4. Number of evaluations required to reach the RMSE of dT(rs) (with 2,500 evaluations) for predicting CPU and DIMM power of lu, mm, and stencil on the test set $\mathcal{T}_{25\%}$.

The results show that the batch mode ab-dynaTree versions are between one to two orders of magnitude faster than the sequential version. The RMSE of the sequential version after 2500 evaluations is worse than that of batch mode versions. On lu (cpu), lu (dimm), mm (dimm) the adoption $n_b = 50$ allow ab-dynaTree to reach lower RMSE than those with 100 and 200. However, on mm (cpu), $n_b = 200$ is better. These observations indicate that there is a tradeoff between n_b and the RMSE: adoption of large n_b allows ab-dynaTree to rapidly reduce the RMSE but since it can perform only few iterations and the final RMSE is worse than others; on the other hand smaller valuers of n_b increases the number of iterations but allows ab-dynaTree to learn the response function effectively.

V. Summary and Outlook

We have proposed an algorithm that adaptively selects inputs for parallel evaluation in order to build efficient surrogate models over the input space. We augmented a popular active learning scheme for the sequential design of experiments to ensure that each simulation node is working on disparate yet useful inputs whose outputs, taken collectively, will lead to updates nearly informative as one-at-a-time schemes previously deployed in serial environments. Our experiments show that this scheme can be effective for building surrogates for empirical autotuning metrics such as run time, power consumption, and FLOPS on a variety of architectures. For the surrogate model types tested, including our preferred dynamic tree model (leading to ab-dynaTree) and comparators such as random forests and neural networks, our active learning approach yields input points that produce better surrogates ones derived from random sampling. Naturally, the significance of this benefit, especially as pertaints to the benefit of one

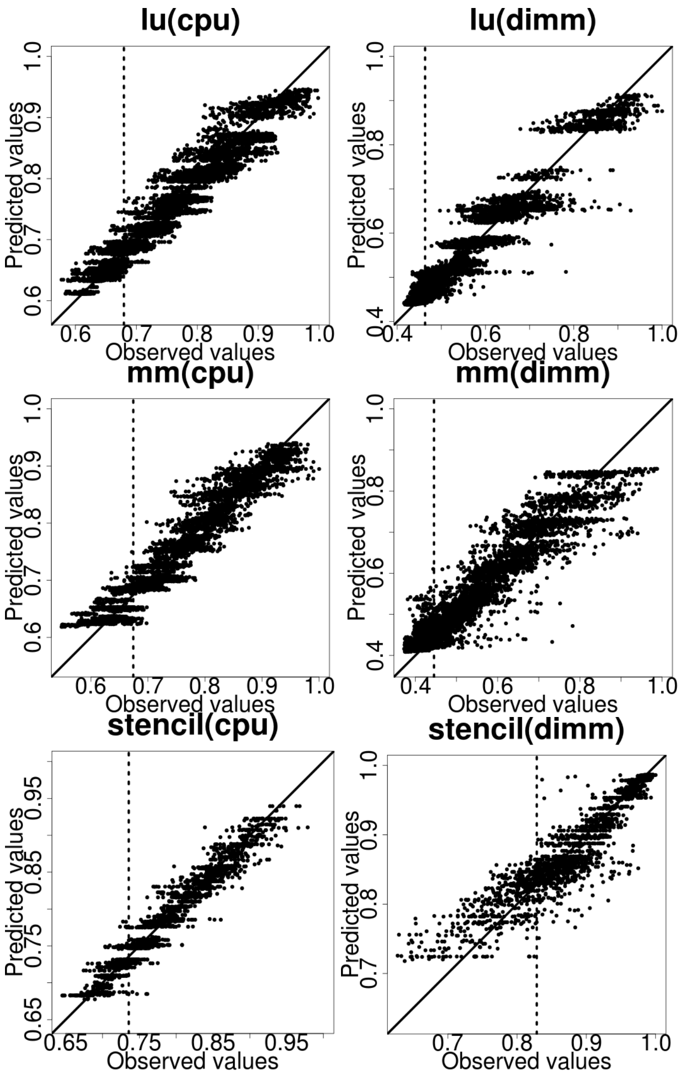

Fig. 5. Correlation plot for the CPU and DIMM power of lu, mm, and stencil on $\mathcal{T}_{100\%}$. The vertical, dotted line represents the 25% quantile.

surrogate model (e.g., `ab-dynaTree`) over others, depends on the output characteristics over the input space.

There are a number of straightforward extensions that would improve the power of `ab-dynaTree` for practical autotuning. For example `ab-dynaTree` can be made completely asynchronous by overwriting $\mathcal{M}_{\text{imp}}$ with $\mathcal{M}$ whenever new data becomes available, rather when the batch has fully been evaluated. Perhaps a scheme similar to that deployed with TGP for a rocket booster experiment [17] would be appropraite. In that work, the authors simply substituted predictive means in for the outputs of running simulations, which is less sophisticated than the a posteriori calculations we propose here. Similarly, it may be advantageous to design several related experiments (e.g., for run time, power consumption, and FLOPS) at once, i.e., having multiple surrogates (one for each) sharing a single design. One option is to deploy a round-robin scheme with independent surrogates [17], which in the batch setting is equivalent to allocated a fixed portion of new runs to be chosen by each surrogate. Although that represents a sensible first step, it ignores the correlation structure between

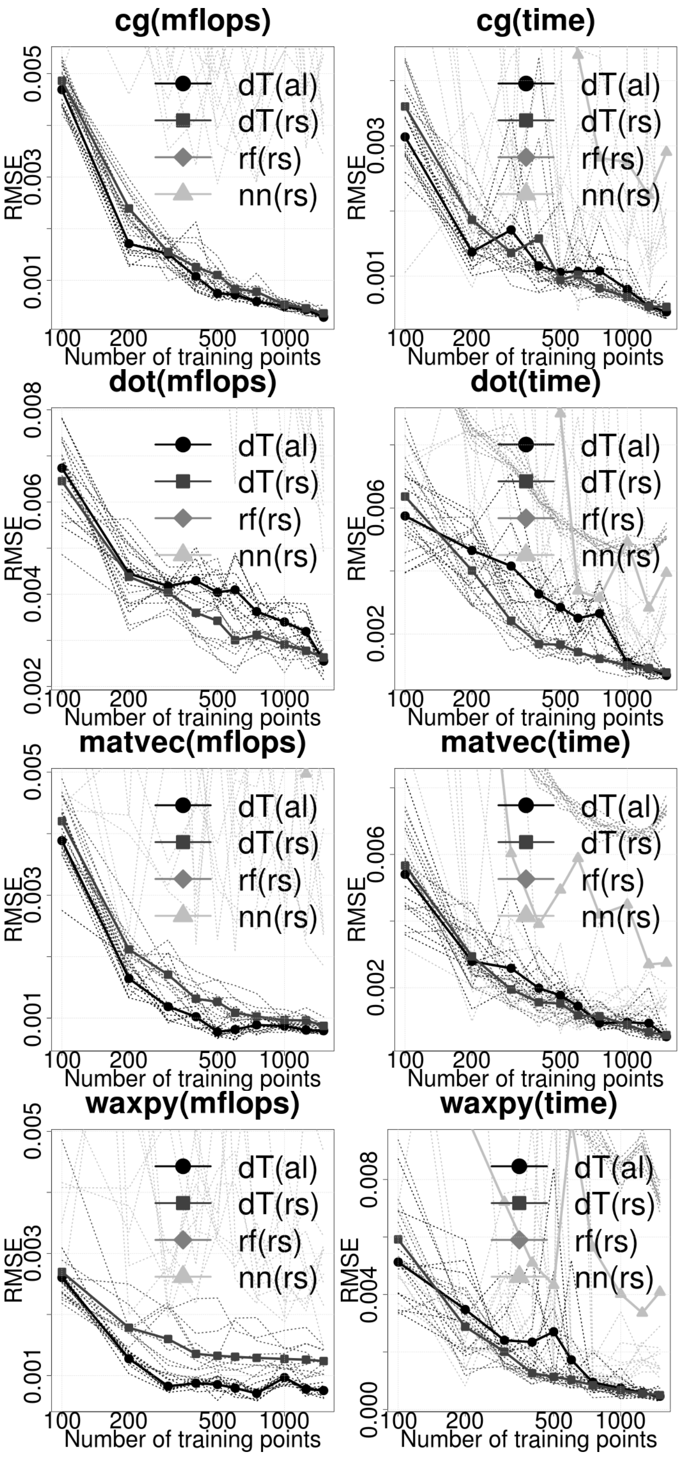

Fig. 6. RMSE for predicting run times and FLOPS of miniFE kernels on the test set $\mathcal{T}_{25\%}$. The dotted lines represent the RMSE for each replication and the bold lines represent the mean RMSE over 10 replications.

obviously related outputs. E.g., power consumption is likely to be negatively correlated with run time. The two-stage model proposed in [26], although it remains to be seen if if extensions for an efficient ALC-like calculation could be derived in that context.

Wasn't sure what to do with these other ones – I'll leave them for one/both of you.

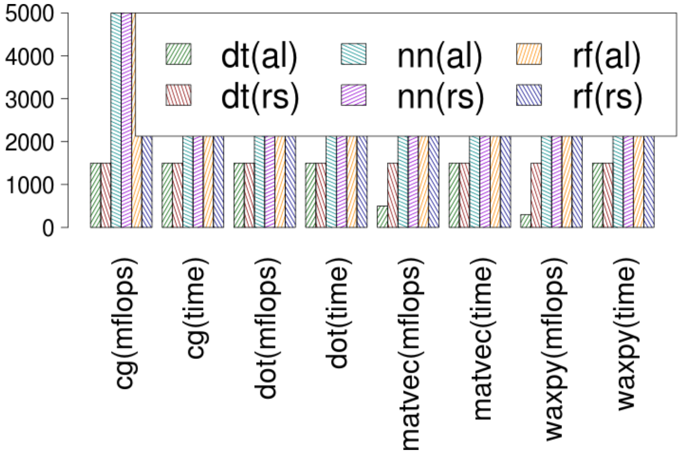

Fig. 7. Number of evaluations required to reach the RMSE of dT(rs) (with 1,500 evaluations) for predicting run times and FLOPS of miniFE kernels on the test set $\mathcal{T}_{25\%}$.

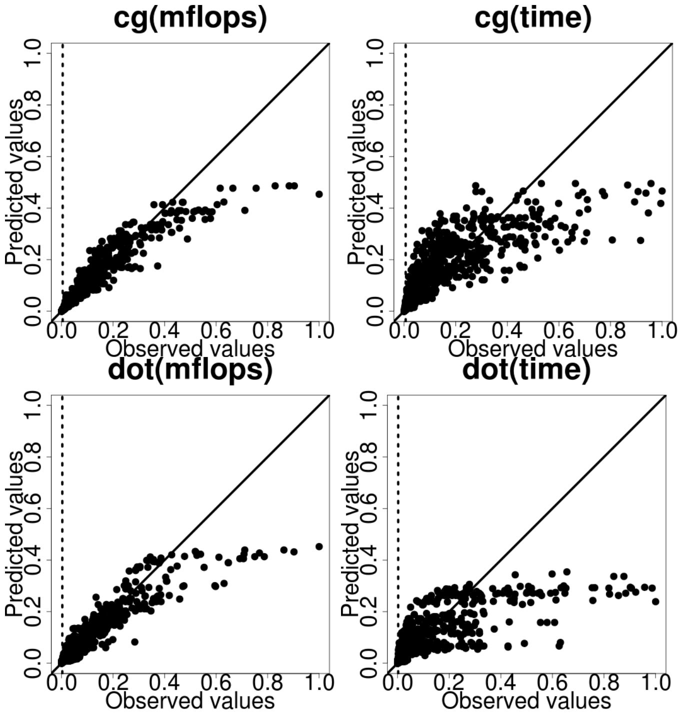

Fig. 8. Correlation plot of observed and predicted values for run times and FLOPS of miniFE kernels on $\mathcal{T}_{100\%}$. The vertical, dotted line represents the 25% quantile.

- `ab-dynaTree` can incorporate existing code variant evaluations to build surrogates with fewer (new) evaluations.
- Capturing/modeling metrics at a finer granularity (e.g., run times of individual kernels or input sizes composing a full-scale application code) to exploit additional structure for whole application modeling and tuning.

Acknowledgments

Support for this work was provided through the Scientific Discovery through Advanced Computing (SciDAC) program funded by the U.S. Department of Energy, Office of Science, Advanced Scientific Computing Research under Contract No. DE-AC02-06CH11357. Computational resources were provided by the Argonne Laboratory Computing Resource Center and the Argonne Leadership Computing Facility. We thank Ananta Tiwari for providing the power measurement data from [5].

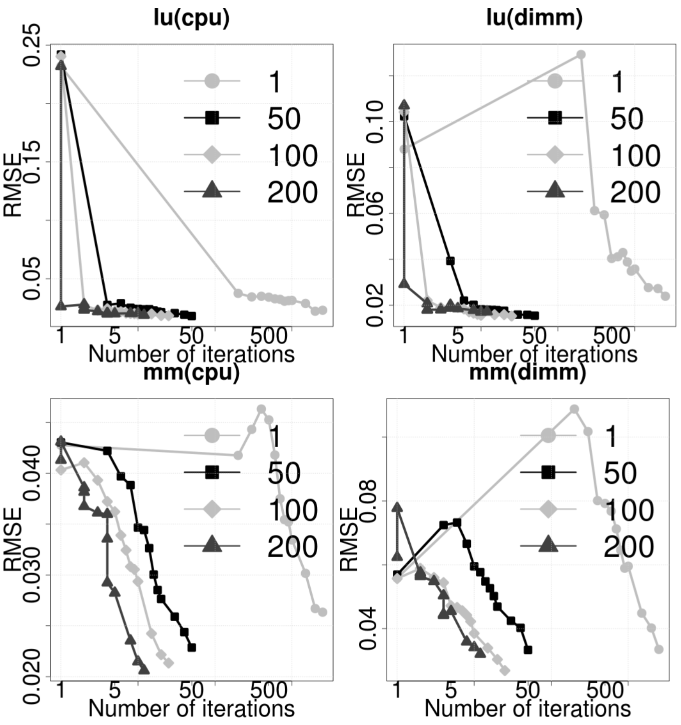

Fig. 9. Impact of batch size on mean RMSE on the test set $\mathcal{T}_{25\%}$.

REFERENCES

[1] D. H. Bailey and A. Snavely, "Performance modeling: Understanding the past and predicting the future," in Euro-Par 2005 Parallel Processing. Springer, 2005, pp. 185–195.

[2] D. Bailey, R. Lucas, and S. Williams, Performance Tuning of Scientific Applications. Chapman & Hall, 2010.

[3] R. Vuduc, J. Demmel, and J. Bilmes, "Statistical models for empirical search-based performance tuning," Int. J. High Perf. Comput. Appl., vol. 18, no. 1, pp. 65–94, 2004.

[4] A. Ganapathi, H. Kuno, U. Dayal, J. L. Wiener, A. Fox, M. Jordan, and D. Patterson, "Predicting multiple metrics for queries: Better decisions enabled by machine learning," in IEEE Int. Conf. Data Engineering (ICDE'09), 2009, pp. 592–603.

[5] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely, "Modeling power and energy usage of HPC kernels," in IEEE Int. Conf. Par. Distrib. Proc. Symp. Workshops (IPDPSW12), 2012, pp. 990–998.

[6] J. Bergstra, N. Pinto, and D. Cox, "Machine learning for predictive autotuning with boosted regression trees," in Innovative Parallel Computing (InPar'12). IEEE, 2012, pp. 1–9.

[7] S. F. Rahman, J. Guo, and Q. Yi, "Automated empirical tuning of scientific codes for performance and power consumption," in ACM Int. Conf. High Perf. Embed. Arch. Comp. (HiPEAC '11), 2011, pp. 107–116.

[8] M. Stephenson, S. Amarasinghe, M. Martin, and U.-M. O'Reilly, "Meta optimization: Improving compiler heuristics with machine learning," ACM SIGPLAN Notices, vol. 38, no. 5, pp. 77–90, 2003.

[9] J. Cavazos, "Intelligent compilers," in IEEE Int. Conf. Cluster Comput., 2008, pp. 360–368.

[10] G. Fursin, C. Miranda, O. Temam, M. Namolaru, E. Yom-Tov, A. Zaks, B. Mendelson, E. Bonilla, J. Thomson, H. Leather et al., "MILEPOST GCC: Machine learning based research compiler," in GCC Summit, 2008.

[11] J. Cavazos, G. Fursin, F. Agakov, E. Bonilla, M. F. O'Boyle, and O. Temam, "Rapidly selecting good compiler optimizations using performance counters," in IEEE Int. Symp. Code Gen. Opt. (CGO'07), 2007, pp. 185–197.

[12] P. Shivam, S. Babu, and J. Chase, "Active and accelerated learning of cost models for optimizing scientific applications," in Int. Conf. Very Large Databases, 2006, pp. 535–546.

[13] R. L. Plackett and J. P. Burman, "The design of optimum multifactorial experiments," Biometrika, vol. 33, no. 4, pp. 305–325, 1946.

[14] T. J. Santner, B. J. Williams, and W. I. Notz, The Design and Analysis of Computer Experiments. Springer, 2003.

[15] M. C. Fu, "Optimization for simulation: Theory vs. practice," INFORMS J. Comput., vol. 14, no. 3, pp. 192–215, 2002.

[16] B. Settles, Active Learning. Morgan & Claypool, 2012.

[17] R. B. Gramacy and H. K. Lee, "Adaptive design and analysis of supercomputer experiments," Technometrics, vol. 51, no. 2, pp. 130–145, 2009.

[18] M. A. Taddy, R. B. Gramacy, and N. G. Polson, "Dynamic trees for learning and design," J. Amer. Statist. Assoc., vol. 106, no. 493, pp. 109–123, 2011.

[19] H. Chipman, E. George, and R. McCulloch, "Bayesian treed models," Mach. Learn., vol. 48, pp. 303–324, 2002.

[20] C. Anagnostopoulos and R. Gramacy, "Dynamic trees for streaming and massive data contexts," The University of Chicago, Tech. Rep., 2012, arXiv:1201.5568.

[21] R. B. Gramacy, M. A. Taddy, and S. M. Wild, "Variable selection and sensitivity analysis via dynamic trees with an application to computer code performance tuning," Ann. App. Stats., vol. 7, pp. 51–80, 2013.

[22] R. B. Gramacy and M. A. Taddy, dynaTree: Dynamic Trees for Learning and Design, 2011, R package version 2.0. [Online]. Available: http://faculty.chicagobooth.edu/robert.gramacy/dynaTree.html

[23] D. A. Cohn, "Neural network exploration using optimal experimental design," in Advances in Neural Information Processing Systems, vol. 6(9). Morgan Kaufmann Publishers, 1996, pp. 679–686.

[24] P. Balaprakash, S. M. Wild, and B. Norris, "SPAPT: Search problems in automatic performance tuning," Proc. Comp. Sci., vol. 9, pp. 1959–1968, 2012.

[25] M. A. Heroux, D. W. Doerer, P. S. Crozier, and J. M. Willenbring, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, September 2009.

[26] S. Conti and A. OHagan, "Bayesian emulation of complex multi-output and dynamic computer models," Journal of Statistical Planning and Inference, vol. 140, pp. 640–651, 2010.