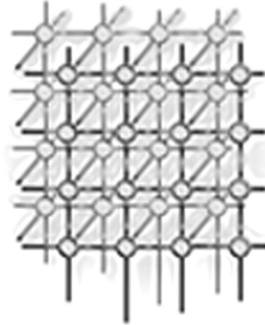


# Predicting parallel application performance via machine learning approaches



Karan Singh<sup>1</sup>, Engin İpek<sup>1</sup>, Sally A. McKee<sup>1,\*</sup>  
Bronis R. de Supinski<sup>2</sup>, Martin Schulz<sup>2</sup>, and Rich Caruana<sup>3</sup>

<sup>1</sup> *Computer Systems Lab, Cornell University, Ithaca, NY 14853*

<sup>2</sup> *Center for Applied Scientific Computing, Lawrence Livermore National Laboratory, Livermore, CA 94551*

<sup>3</sup> *Computer Science Department, Cornell University, Ithaca, NY 14853*

---

## SUMMARY

Consistently growing architectural complexity and machine scales make creating accurate performance models for large-scale applications increasingly challenging. Traditional analytic models are difficult and time-consuming to construct, and are often unable to capture full system and application complexity. To address these challenges, we automatically build models based on execution samples. We use multilayer neural networks, since they can represent arbitrary functions and handle noisy inputs robustly. In this paper we focus on two well known parallel applications whose variations in execution times are not well understood: SMG 2000, a semicoarsening multigrid solver, and HPL, an open source implementation of LINPACK. We sparsely sample performance data on two radically different platforms across large, multi-dimensional parameter spaces and show that our models based on this data can predict performance within 2% to 7% of actual application runtimes.

KEY WORDS: high performance computing; performance modeling; artificial neural networks

---

\*Correspondence to: Sally A. McKee, 324 Rhodes Hall, Electrical and Computer Engineering, Cornell University, Ithaca, NY 14853, USA

Contract/grant sponsor: National Science Foundation; contract/grant number: CCF-0444413

Contract/grant sponsor: United States Department of Energy; contract/grant number: W-7405-Eng-48

\*The submitted manuscript has been co-authored by contractors of the U.S. Government under contract number W-7405-Eng-48. Accordingly, the U.S. Government retains a nonexclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

---



---

## INTRODUCTION

Both architecture and software complexity are rising dramatically, and their interactions are often unclear to both developers and users. Furthermore, parameter spaces of interest are growing for most high-end applications. As a direct consequence, creating accurate models for modern systems and applications becomes increasingly difficult and time consuming. Under these circumstances, the traditional approach to analytical modeling often fails. Construction of the models is a long and error-prone process requiring detailed understanding of target systems and applications (knowledge that is increasingly difficult to acquire). Further, analytical models necessarily make simplifying assumptions for both the target system and the input space, leading to loss in accuracy and generality as well as failure to capture subtle interactions between architecture and software.

Instead, we can leverage machine learning, using results from a subset of executions from the full application parameter spaces as samples from which to construct models for the remaining parameter spaces. Several techniques exist for this kind of approach, including various regression methods and Support Vector Machines. We choose neural networks because they are a well studied and robust technique, and they allow the representation of any functional model without a priori specifications. Neural networks have been shown to work well even when the samples contain noise (a particular problem in our chosen arena of application), and they generally require small training sets to construct the models. The latter translates to smaller numbers of samples and hence—in our case—fewer application executions. The next section explains network training in detail.

In this paper, we demonstrate how we construct and use our neural network models, and we identify two challenges of our approach to performance prediction: handling noise in the dataset, and choosing appropriate techniques for the training phases of the neural networks. We experiment with different techniques to construct the sample sets on which our models are trained and with ensemble learning mechanisms to decrease variance. We use two well known numerical benchmark codes, SMG2000 [8] from the ASCI Purple benchmark suite [13] and the High Performance LINPACK (HPL) [16] implementation from the University of Tennessee, and study their performance across a range of input parameters. The resulting models can predict performance across large, multi-dimensional parameter spaces on two large-scale parallel platforms within 2% to 7% error. We find our approach to be useful for many application performance prediction problems [10], and our techniques are particularly well suited to mining performance databases or to extending fast, parameter-specific models.

## APPROACH

We generate models to predict application performance across a large, multidimensional parameter space defined by program inputs. To capture all complex interactions between the target architecture and software, we create a sample set of this parameter space by executing the target application on real hardware and gathering the resulting performance data. We then use machine learning techniques to automatically train corresponding models to cover the complete input space. Although there are other predictive modeling methods (such as



linear or polynomial regression, Support Vector Machines [SVMs], and decision trees), we base our approach on Artificial Neural Networks (ANNs) because they:

- constitute a mature and commercialized technology. They represent one of the most powerful and flexible methods known for performing generalized nonlinear regression.
- do not require that the form of the functional relationship between inputs and target values be known. Their representational power is rich enough to capture complex, nonlinear interactions between multiple variables. Any function can be approximated to arbitrary precision by an ANN with three layers of units.
- work well with noisy data. Samples generated by executions of the application on real hardware are naturally noisy, and a predictive model must perform well despite slight inaccuracies in the samples used to build it.

For all experiments, we first select a representative dataset by choosing a collection of points spread across the complete parameter space; we then obtain performance results for these on actual hardware. We reserve a portion of this data as a *test* set against which to report the final accuracy of our models, and never train on this test data. From the remaining sampled data, we choose a subset of points and use these data to build our models. During this process, we split the data into separate *training* and *validation* sets, where the former is used to actually train the neural network model, and the latter is used to assess the error of the current model at each step during training. After training, we query the final model to obtain predictions for points in the full parameter space, and report the accuracy of our model on points included in the test set.

### Target Applications and Their Characteristics

We study two well known numerical applications: SMG2000, a semicoarsening multigrid solver based on the *hypr* library [8]; and HPL, a portable implementation of High Performance LINPACK [16] used to solve (random) dense linear systems on distributed memory computers.

Our goal is to predict application runtimes to assist in resource usage estimation, to contribute to understanding of application behavior, and to aid in tuning input and algorithm parameters. However, for both applications the performance variations for different input parameters and the interactions with a given target system are not well understood. We develop application-specific performance models for these applications on two radically different parallel architectures. This enables predicting runtime or other important characteristics across large input parameter spaces with high dimensionality.

SMG2000 has a six-dimensional parameter space that describes both the shape of the workload per processor and the logical processor topology. These parameters have substantial impact on runtime, as shown in Figure 1. For a fixed working set size—a fixed subvolume size per CPU—runtime varies by up to a factor of five. This application employs a complex, recursive algorithm to decompose its three dimensional grid making predictions difficult. Consequently, only a rough analytical model approximating the communication volume for cubic working set sizes has been built to date [4]. While it would be possible to extend this model for arbitrarily shaped working sets, doing so would be extremely complex, and

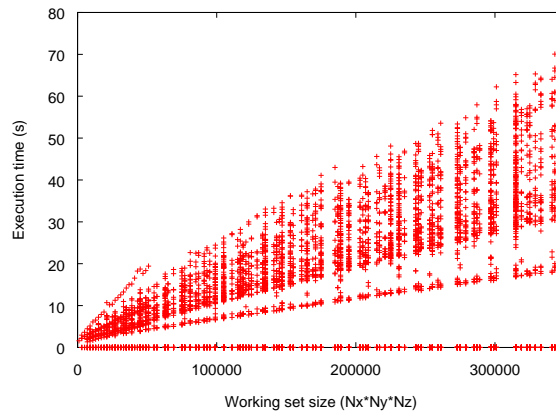


Figure 1. Execution times for SMG2000 for varying processor workloads  $(N_x, N_y, N_z)$  and processor topologies  $(P_x, P_y, P_z)$  running on 512 Blue Gene/L nodes

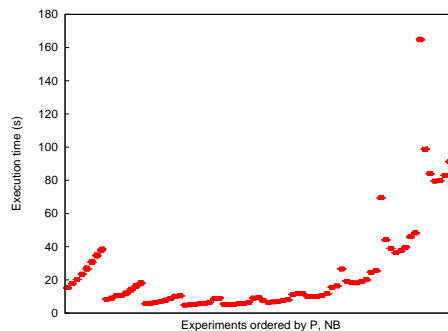


Figure 2. Execution times for HPL with varying block sizes  $(NB)$  and topologies  $(P)$  running on 512 Blue Gene/L nodes

the result would likely be intractable. Worse, modeling architectural features in addition to overall performance is infeasible. Our automatic, empirical modeling approach overcomes these limitations *without knowledge of the application or its algorithms*.

The HPL solver uses a two-dimensional, block-cyclic data distribution and LU factorization with row partial pivoting featuring multiple look-ahead depths. Recursive panel factorization with pivot search and column broadcast via MPI combined with a bandwidth-reducing swap broadcast approach make the package scalable with respect to parallel efficiency for a given



per-processor memory utilization. Details of the algorithm can be fine-tuned with a large range of parameters. However, only rough guidelines exist on how to choose the best settings for a given target architecture [16], forcing the user to rely on hand-tuning for each platform. We choose five parameters that are generally known to be most significant, and we vary processor topology in tandem. Again, we observe that application execution times change dramatically with varying parameters, as shown in Figure 2.

## Neural Networks

Artificial Neural Networks (ANNs) are a class of machine learning models that map a set of input parameters to a set of target values. A neural network is composed of a set of *units* that process the values at their inputs to produce a single scalar value. The set of weighted incoming edges at each unit indicates the set of values communicated to it. Every unit forms a sum of these values multiplied by the weights associated with corresponding edges. The weighted sum is input to an *activation function* that produces the unit's output.

Figure 3 shows an example of a neural network architecture. Each node represents a unit and each edge a weighted connection between two units. In this example network, input parameters are placed at the first (lowest) layer, and information flows from bottom to top. The units producing the final predictions are *output units*, and those that receive input parameters are *input units* (input units simply pass incoming values to all their outgoing edges). In addition, one or more layers of *hidden units* may be part of the network architecture. Hidden units process outputs of other units, and pass their own outputs to another set of (hidden or output) units. The representational power of a neural network (the set of functions it can represent) can be increased by adding hidden units and layers. When every unit in a layer receives values from all units in the layer below, the neural network has a *fully connected feedforward* architecture. Figure 3's example is such an ANN with three input units, one output unit, and a single layer of four hidden units.

We use fully connected feedforward neural networks with sigmoid activation functions. Activation functions need not be sigmoid, but must be non-linear, monotonic, and differentiable. After a prediction on the current example, the weights in the network are updated in proportion to their contribution to the error. Note that our models only predict application runtimes, but ANNs are not restricted to a single output (and in fact perform better when predicting multiple outputs).

The weights associated with edges in an ANN define the functional relationship between input and output values, and they are set during the training phase of building the model. Training an ANN is an iterative process that involves learning edge weights from a set of sample data points (tuples of input and output values). We use *rprop* (*resilient backpropagation*) to train edge weights [17]. In general, backpropagation approaches use gradient descent in weight space to minimize the squared error between simulation results and model predictions. Edge weights are initialized near zero, causing the network to act like a simple linear model at first. Here we initialize weights uniformly on  $[-0.01, +0.01]$ . During training, examples are repeatedly presented at the inputs, differences between network outputs and target values are calculated, and backpropagation updates all weights by taking a small step in the direction of steepest decrease in error. As the weights grow, the ANN becomes increasingly non-linear. Standard

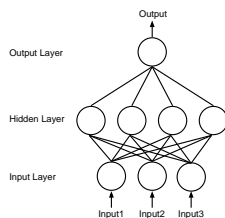


Figure 3. A feedforward neural network with a single hidden layer

backpropagation updates weights using the squared error and a small *learning rate* constant (effectively the gradient descent step size). Rprop is a locally adaptive training algorithm that only propagates the sign of the error such that a unique update rule evolves for each weight. Models using rprop tend to learn more quickly, and the algorithm runs faster than standard backpropagation.

### Network Refinements

We first apply standard feedforward neural networks to our problem domain. This naive approach achieves poor accuracy for two main reasons. First, system activities sharing resources with application threads create nondeterministic variations in performance, yielding significant noise in the dataset. Accuracy on future runs can never exceed this noise level. Second, backpropagation training algorithms for neural networks are generally designed to reduce absolute mean-squared-error, but are unsuitable for reducing percentage error. During training, examples on which the model makes higher absolute error are given greater weight when we use the naive approach, even though the error may be small in relative terms. For instance, given test cases  $t_1$  with a runtime of 100 seconds and  $t_2$  with a runtime of one second, an error of 0.5 seconds would be given equal weight for both, even though the percentage error varies drastically between the two examples (0.5% vs. 50%).

Applying ANNs to application performance prediction requires both a mechanism for reducing noise during data collection and a technique to train the networks for percentage error. Reducing the noise level dictates that the difference between performance results from two different runs with the same input parameters be kept as small as possible. On certain computing platforms where operating system activity is minimal (e.g, Blue Gene/L), this problem is negligible. On platforms with full-featured OS instances on each node (including Linux clusters and traditional large scale MPPs), operating system interference (even on dedicated partitions) can lead to significant noise levels. To avoid this, we must choose a system setup that minimizes noise, and we find that reserving at least one processor per node for system processing greatly reduces the problem.

Once noise levels are acceptably diminished, a mechanism for training the neural network to reduce percentage error is needed. For this we use techniques that steer the training set to



emphasize samples with low absolute but high relative error. Here we study *stratification* and *active learning* [18]. In addition, we apply the ensemble learning techniques *bagging* (bootstrap aggregation) [3] and cross validation. These techniques create sets (ensembles) of models from subsets of samples and then create a final model by averaging all models within the ensemble.

Our previous work [9] studies just the SMG datasets, using standard backpropagation with stratification and bagging (described below) to build our models. Results in the next section represent a much deeper exploration of applying ANNs to parallel program performance prediction.

### *Stratification*

Stratification allows us to put varying amounts of emphasis on different regions of the search space by replicating sample points that are expected to have higher error rates. The model is trained more often on these values, correcting for the expected error distribution. We apply stratification to reduce percentage error, and hence we replicate each point in the dataset by a factor proportional to the inverse of its target value. As a consequence, the network is being trained more on points with small target values, which have large relative but low absolute errors. As a consequence, the absolute error is reduced for these samples to the point that the relative errors across the whole model no longer show large divergences. Stratification has the nice property that the number of samples does not increase, since only existing points are replicated within the training set. This allows us to apply this technique without having gather more data, which can be costly. Our prior work [9] finds that stratification significantly improves the performance of standard backpropagation.

### *Ensemble Learning Techniques*

An ANN with a large enough hidden layer can approximate any continuous function. As in polynomial curve fitting, where using a polynomial of high degree results in models that have excellent fit to the training samples yet interpolate poorly, ANNs also may *overfit* to the training samples. However, unlike polynomial curve fitting, where model complexity is reduced by decreasing the degree of the polynomial, large ANNs with excess capacity usually make better predictions when training is halted before gradient descent reaches the minimum error on the training set [15, 7]. A portion of the training set (the *early stopping set*) is therefore reserved, and gradient descent stops when the model's squared error on this unbiased sample stops improving.

Holding aside a fraction of the training data to determine when to halt training is an effective way to prevent overfitting in ANNs. Unfortunately, if 25% of the data is used as the early stopping set, the training set used for gradient descent is 25% smaller. As with other regression methods, ANNs learn less accurate models when training set size is reduced. Ensemble learning techniques avoid this problem by creating multiple models for each sample set by extracting multiple overlapping subsets and creating individual models for each using (a part of) the remaining points outside the early stopping set. After all models are created, they are combined into a single average model that then contains information from all points



present in at least one of the subsets. In addition, ensemble techniques such as *cross validation* and *bootstrap aggregation (bagging)* allow us to estimate model accuracy, as explained below.

In cross validation, the training sample is split into multiple equal subsets, or *folds*. For example, 10-fold cross validation splits the training sample into 10 equal-sized folds, each containing 10% of the training data. An ANN is then trained on the samples in folds 1-8 (80% of the data); fold 9 (10% of the data) is used for early stopping; and fold 10 (also 10% of the data) is used to estimate the performance of the trained model. A second ANN is trained on folds 2-9, fold 10 is used for early stopping, and fold 1 is used to estimate accuracy. This process is repeated 10 times, with the data in each fold being used successively as early stopping sets and test sets. The 10 networks that result from 10-fold cross validation are combined into an ensemble by averaging the predictions made by each ANN. Although each ANN is trained on only 80% of the training data, all data are eventually used to train some models in the final ensemble. The ensemble thus performs similarly to a model trained on all data, yet reserved data is always available for early stopping and unbiased error estimation. Experience has shown that averaging multiple models (an approach frequently used in weather forecasting) often delivers better performance than training just one model. The mean and standard deviation of the model errors made on the 10 test folds are used to estimate the accuracy of the cross validation ensemble, allowing the user to determine when the models are sufficiently accurate. In general, partitioning the data into more folds results in lower error rates and better estimates of network accuracy, at the expense of a slightly higher computational cost to train more models.

*Bagging* is a general machine learning method that generates an ensemble of models by training them on bootstrap samples that are generated by randomly drawing  $N$  instances with replacement from the training set. A single instance from the training set has probability  $1-(1-1/N)^N$  of being included at least once in any bootstrap sample. In the limit, the probability for each instance is  $(1-1/e)$ , and thus a single bootstrap sample contains about 63% unique instances from the training set. The differences among bootstrap samples generate different models. If the resulting models in the ensemble are uncorrelated, they can improve performance for unstable learning algorithms like ANNs. As with cross validation, the final prediction is derived from averaging predictions from all models in the ensemble. Likewise, averaging multiple models via bagging often yields better performance than training just one model.

## Active Learning

*Active learning* is a general class of algorithms that aim for a given accuracy using the smallest possible sample set. The approach tries to select points from which the model is likely to derive the most benefit. We seek to identify samples on which the model makes greatest error, since learning these points is most likely to improve model accuracy. We are faced with a “chicken and egg” problem, though: assessing model error on any point requires knowing results for that point, and the application execution to generate that data point has not yet been run. We therefore combine active learning with the ensemble learning techniques mentioned above, and use the predictions of multiple models to approximate the error for each point. After each training round, we query all ANNs for predictions on every point in the parameter space (testing takes a few seconds). We calculate the variance of model predictions,



	Blue Gene/L	Thunder
Processor	PowerPC 440	Intel Itanium 2
Frequency	700MHz	1.4GHz
L1 ICache	32KB	32KB
L1 DCache	32KB	32KB
L2 Cache	2KB (Prefetch Buffer)	256KB
L3 Cache	4MB	4MB
SDRAM	512MB DDR 350	8GB DDR 266
Network	3D Torus + Global Combine/Broadcast Tree Network	Fat Tree (Quadrics QsNet)
Tasks/Processors per node	1/2	3/4
Number of Nodes Used	512	64

Table I. Platform parameters

the mean prediction, and the prediction's coefficient of variance, or  $CoV$  (the ratio of the standard deviation to the mean). The ANNs in the ensemble disagree most on points with high prediction  $CoV$  values. Since the ANNs differ primarily by the samples (folds) on which they are trained, high disagreement indicates that including the point in the sample set can lower model variance (and thus error). Note that impacts of different points can be dependent on one another, and simply sorting and sampling based on  $CoV$  values can yield a dataset with redundant points. We iteratively choose samples from the sorted points, first including the least confident point, then the next least confident point whose distance in the parameter space from the first point is above a certain threshold, and so on. If too few points satisfy the distance constraint, we lower the threshold and reconsider previously rejected points. Once we have enough sample points, we train a new model; if our error estimate is still too high, we calculate a new set of points for sampling through the active learning mechanism, and repeat.

## EXPERIMENTS

We present performance prediction results for SMG2000 and HPL on two large-scale machines at the Lawrence Livermore National Laboratory: Thunder, a 1024 node Linux cluster with Itanium-2 processors, and Blue Gene/L, a tightly integrated system with 65536 compute nodes. Table I gives details of these platforms. Nodes on Thunder are four-way SMPs, and as noted above, we run only three tasks per node to reduce noise caused by OS interference. Nodes on Blue Gene/L have a single compute ASIC with two embedded Power 440 cores. We use one task per node in "communication coprocessor" mode: one core performs main computation, while the other is dedicated to networking operations [2].

For both SMG2000 and HPL we explore a six-dimensional parameter space. Table II and Table III show the parameters and their possible values for the SMG2000 and HPL codes, respectively. For both applications we choose the default values for all other parameters. Note that for SMG  $N$  describes the size of a processor's *local* working set, whereas for HPL  $N$



Parameter	Blue Gene/L	Thunder
$N_x$	10-510 in steps of 20	10-250 in steps of 30
$N_y$	10-510 in steps of 20	10-250 in steps of 30
$N_z$	10-510 in steps of 20	10-250 in steps of 30
$P_x$	1,8,64,512	1,3,4,12,16,48,64,192
$P_y$	1,8,64,512	1,3,4,12,16,48,64,192
$P_z$	1,8,64,512	1,3,4,12,16,48,64,192
Constraints	Blue Gene/L	Thunder
$P_x * P_y * P_z$	512	192
$N_x * N_y * N_z$	$1000 \leq N_x * N_y * N_z \leq 343000$	$216000 \leq N_x * N_y * N_z \leq 9261000$

Table II. SMG2000 application parameters and constraints:  $N_x, N_y, N_z$  describe the size of the three dimensional volume used as the working set per processor;  $P_x, P_y, P_z$  describes the processor topology in all three dimensions; the problem size for a particular run is a volume of size  $N_x * P_y * N_y * P_y \times N_z * P_z$ .

Parameter	Values
$N$ (problem size)	10000
$NB$ (block size)	10-80, stepped by 10
$P \times Q$ (processor grid)	$P = 2^n, Q = 2^{9-n}, 0 \leq n \leq 9$
$PFACT$	R, C, L
$NBMIN$	1, 2, 4, 8
$NDIV$	2, 3
$RFACT$	R, C, L

Table III. HPL application parameters: the total problem size  $N$  is kept constant and we only vary the processor grid topology  $P \times Q$  as well as algorithm parameters— $NB$  controls the blocking size and  $PFACT, NBMIN, NDIV,$  and  $RFACT$  control the recursion depth and data granularity of the solver (for details see [16]).

describes the *global* problem size. The total dataset for SMG2000 on Thunder consists of 6170 points, and on Blue Gene/L 3358 points. The HPL dataset on Blue Gene/L consists of 5760 points. Table IV characterizes the performance of each of the datasets, and again shows the wide spread of possible performance results. Note that these datasets are already sparse, since we only sample linear values at regular intervals. This is especially true for SMG2000:  $N_x, N_y,$  and  $N_z$  are taken in large steps to reduce the number of experiments. The total size of the parameter space is significantly larger. We report sample size percentages in relation to our sparse representation of the full parameter space, but our models can interpolate to predict performance for intermediate values.



dataset	minimum	maximum	mean	stdev
HPL BG/L	4.8097	165.234	24.2629	28.4625
SMG2000 BG/L	1.3527	70.0639	23.6603	13.1221
SMG2000 Thunder	11.1222	5474.5500	81.5215	119.6170

Table IV. Runtime statistics for each dataset

dataset	training set size	stratification cross validation		active learning cross validation		stratification bagging	
		error	stdev	error	stdev	error	stdev
		SMG2000 on Thunder	5%	9.24	38.37	6.98	7.69
	10%	9.01	27.58	5.98	6.85	14.80	16.35
	20%	5.66	6.23	5.47	6.37	6.03	6.54
SMG2000 on BG/L	5%	8.12	6.78	9.44	8.96	9.21	8.17
	10%	6.70	5.99	9.44	8.33	7.33	6.80
	20%	6.22	6.47	7.94	7.47	5.70	5.53
HPL on BGL	5%	4.68	5.23	7.11	9.44	5.85	4.88
	10%	2.86	3.31	4.75	7.58	3.35	3.63
	20%	2.09	2.46	4.76	6.28	2.38	2.40

Table V. Mean errors and standard deviations as training set size increases

We iteratively train and test our models, incrementing the sample dataset by 50 points at each round. For experiments using bagging, instead of ten-fold cross validation, we train on the “in-bag” random samples [3], and use “out-of-bag” samples for early stopping. We randomly select a test set of 1K points from remaining data and report model accuracy on this set.

Even when using rprop over standard backpropagation, models trained only with random sampling versus stratification, active learning, and ensemble techniques are less accurate and learn more slowly, and thus we omit those results here. Figure 4 through Figure 6 show learning curves for mean prediction error (left column), and standard deviations for those prediction errors (right column). The top graphs in each figure show results when we use rprop with 10-fold cross validation for training, and stratification to select sample points. The middle graphs show results for the combination of rprop and cross validation when we use active learning to select training samples. The bottom graphs show results for the combination of rprop and stratification when we use bagging to reduce variance. Predicted values for mean error and standard deviation are derived from the multiple models in our ensemble methods, and actual values show comparisons of model predictions against the full dataset.

Table V gives prediction error and standard deviation for training sets constituting 5%, 10%, and 20% of their respective datasets. The learning curves and the table show how model accuracy improves as training set size increases. For instance, at a training set size of 350 points (approximately 5% of the entire dataset) for SMG2000 running on Thunder (Figure 4), the average errors on the test set are 9.24% with stratification and cross validation and 16.95% with stratification and bagging. The corresponding standard deviations of these errors are

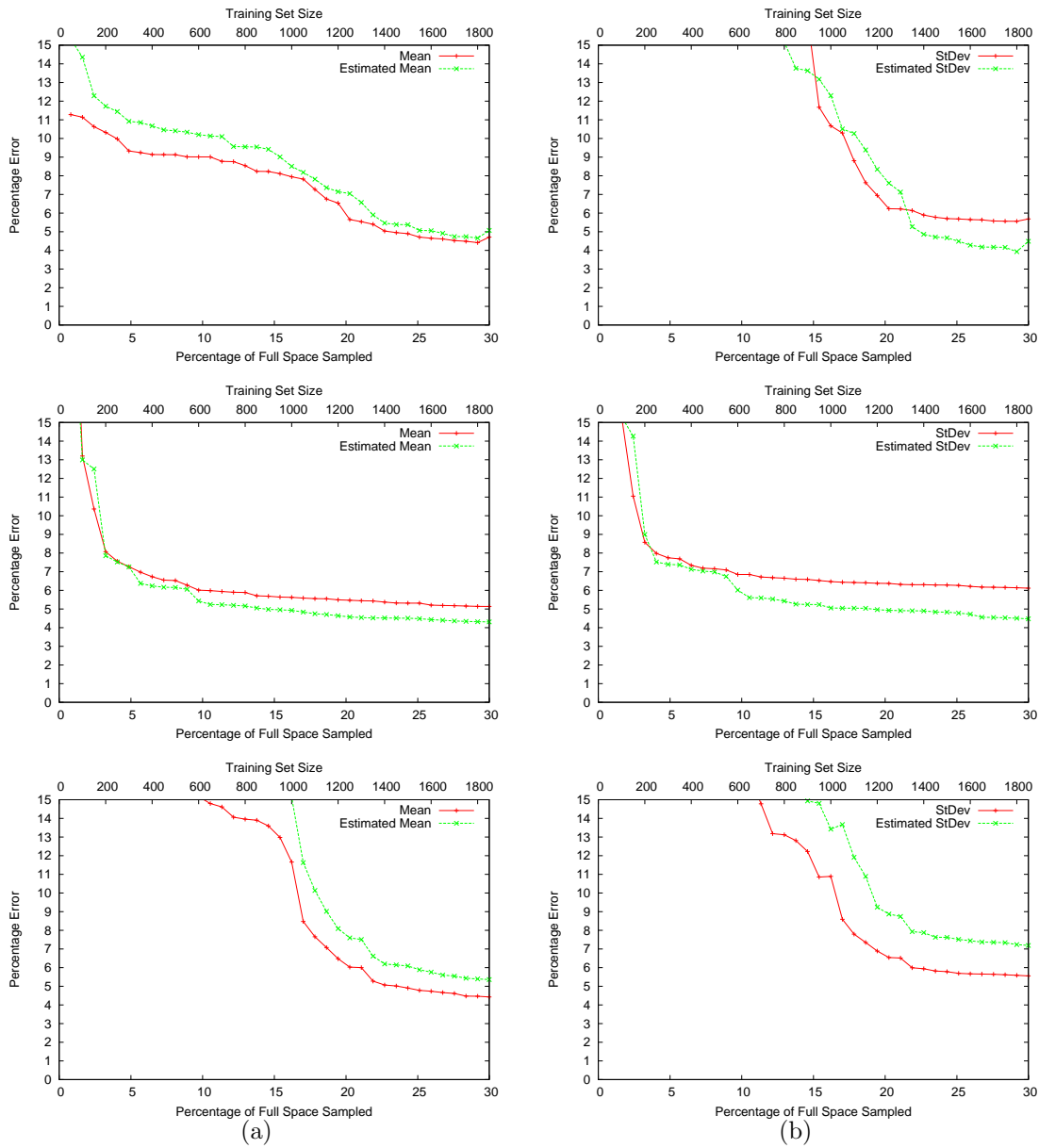


Figure 4. Results for SMG2000 on Thunder: mean predictions (a) and standard deviations (b) for stratification & cross validation (top), active learning & cross validation (middle), and stratification & bagging (bottom)

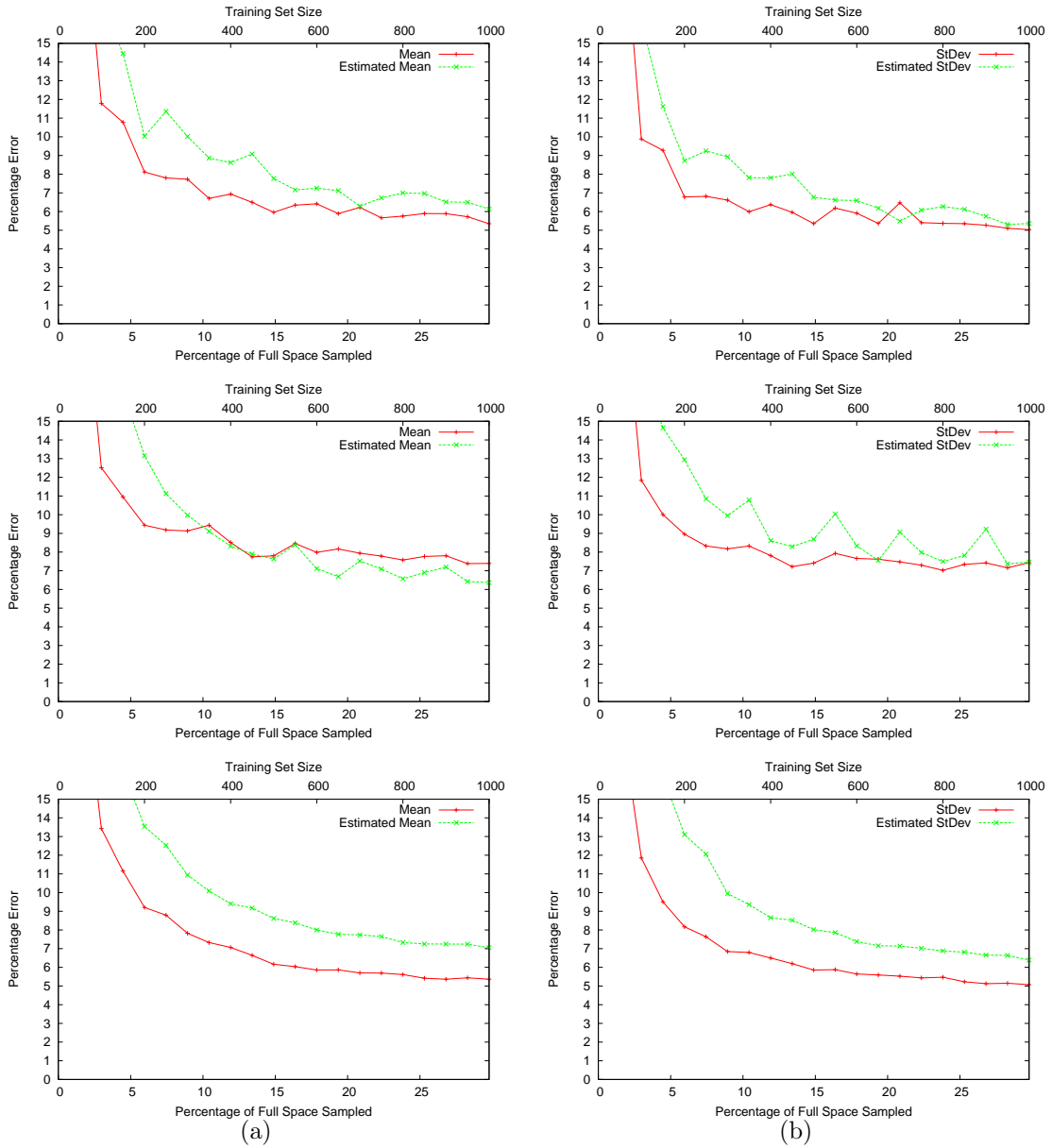


Figure 5. Results for SMG2000 on Blue Gene/L: mean predictions (a) and standard deviations (b) for stratification & cross validation (top), active learning & cross validation (middle), and stratification & bagging (bottom)

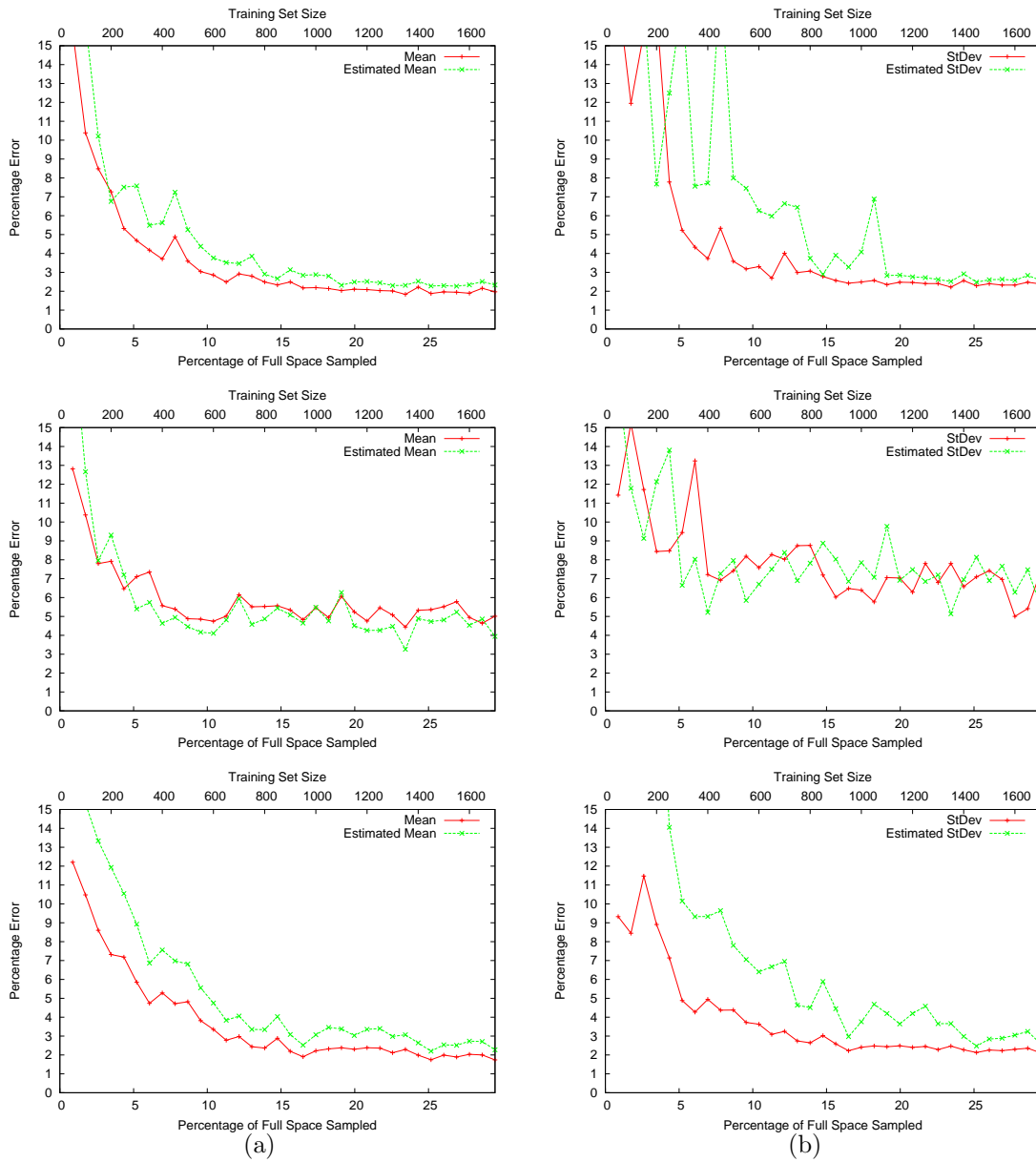


Figure 6. Results for HPL on Blue Gene/L: mean predictions (a) and standard deviations (b) for stratification & cross validation (top), active learning & cross validation (middle), and stratification & bagging (bottom)



38.37% and 26.08%. In the absence of active learning, these training sets are too small and contain too little information to build highly accurate models. Active learning achieves much better results than our other sampling methods on our smaller, noisy datasets from Thunder. The data taken on Blue Gene/L are less noisy than those from Thunder, and hence prediction accuracies are generally higher. As training set size increases, error decreases, showing that the model benefits from the additional information included in the dataset at each round of training, and that choice of sampling method becomes less important as the models are trained on more data. Eventually, the curves begin to flatten, since additional data presented to the models contain little new information. Similarly, the standard deviation of the error decreases with increasing training set size.

No single approach performs best on all datasets. For instance, active learning and cross validation works well for SMG2000 on Thunder at small dataset sizes, but performs worse than stratification and cross validation on the Blue Gene/L datasets (improvements to our active learning approach are part of ongoing work, since intelligent sampling should be able to outperform random selection). Given sufficient training data, bagging outperforms cross validation for SMG, but not for HPL on this platform: the HPL dataset does not exhibit the high variance that makes bagging effective. Stratification with cross validation provides the most robust performance, delivering comparable results to the other two approaches at large training set sizes, and outperforming them at smaller training set sizes (with the exception of active learning on the Thunder dataset). The HPL dataset is the most predictable: our models predict performance with only 2-3% error and similar standard deviations. Ongoing work examines the combination of rprop, stratification, active learning, and cross validation, but space limitations prevent including that data.

These results indicate that the accuracy of our approach can be high given enough training points. Our parameter spaces are much larger than the total number of points we collect. When training on only 30% of our datasets, our models achieve accuracies of about 94-95% on SMG2000 and 95-98% on HPL. Our approach can thus easily be used to learn from performance databases of results for sparse samplings of parameters. In addition, the amount of time required to fully train a model ranges from 1-15 minutes on a typical workstation with a 3.0GHz Pentium 4 processor and 1GB of main memory, making it easy to build parameterized performance models efficiently.

## RELATED WORK

Many prior studies address performance prediction for parallel programs. Most performance modeling techniques require either in-depth knowledge of the applications to build analytical models or special tools to gather such information from parallel codes. Often such approaches are application specific, restricted to a given language, or based on simulation. Nonetheless, with careful modeling of applications and platforms, these approaches can deliver high prediction accuracy.

Marin and Mellor-Crummey [14] semi-automatically measure and model program characteristics, using properties of the architecture, properties of the binary, and application inputs to predict application behavior. Their toolkit predefines a set of functions, and the



user may add customized functions to this library if needed. They vary the input size in only one dimension (in contrast to our studies), and they cannot account for some important architectural parameters (e.g., cache associativity in their memory reuse modeling).

Carrington et al. [5] demonstrate a framework for predicting performance of scientific applications on LINPACK and an ocean modeling application. Their automated approach relies on a convolution method representing a computational mapping of an application signature onto a machine profile. Simple benchmark probes create machine profiles, and a separate tool generates application signatures. Extending the convolution method allows them to model full-scale HPC applications [6]. They require generating several traces, but deliver predictions with error rates between 4.6% and 8.4%, depending on the sampling rates of the underlying traces. Using full traces obviously performs best, but such trace generation can slow application execution by almost three orders of magnitude. Some applications demonstrate better predictability than others, and for these trace reduction techniques work well: prediction errors range from 0.1 to 8.7% on different platforms. This work complements ours, and the two approaches may work well in combination. Their analytic models could provide bootstrap data, and our models could give them full application input parameter generality.

Kerbyson et al. [12] present an accurate, predictive analytical model that encompasses the performance and scaling characteristics of SAGE, a multidimensional hydrodynamics code with adaptive mesh refinement. Inputs to their parametric model come from machine performance information, such as latency and bandwidth, along with application characteristics, such as problem size and decomposition (as in our models). They validate prediction accuracy of the model against measurements on two large-scale ASCI systems. In addition to predicting performance, their model can yield insight into performance bottlenecks, but the application-centric approach requires that the code be statically analyzed, and a separate, detailed model must be developed for each target application. In contrast, our approach is application agnostic and easily automated.

Yang et al. [19] develop cross-platform performance translation based on relative performance between the target platforms, and they do so without program modeling, code analysis, or architectural simulation. Like ours, their method targets performance prediction for resource usage estimation. They observe relative performance through partial execution of two ASCI Purple applications [13]; the approach works well for iterative parallel codes that behave predictably (achieving prediction errors of 2% or lower) and enjoys low overhead costs. Prediction error varies much more widely (from 5% to 37%) for applications with variable overhead per timestep. Likewise, reusing partial execution results for different problem sizes and degrees of parallelization renders their model less accurate.

## CONCLUSIONS AND FUTURE WORK

We have presented a machine learning approach to application performance prediction using multilayer neural networks. We use performance data gathered from real application executions on a small subset of our parameter spaces to train performance models covering the complete parameter space. We also refined and adapted this approach to learn more accurate models in the presence of large runtime difference as well as to reduce the number of required samples.



Our techniques yield highly accurate results for two parallel applications—SMG2000 and HPL—on two different high-performance platforms, delivering prediction error rates of 2% to 7%.

We find our approach to be robust, even in the face of rapidly increasing machine and system complexity and scale. The modeling is easily automated and application agnostic: users need only state relevant input and output parameters and provide access to performance samples, either by executing the target code or using precomputed values. This makes the approach more attractive for our purposes than traditional analytic modeling techniques: we trade the depth of application understanding users gain in developing analytic models for speed, accuracy, and ease of use. Future work will mine existing (and growing ) performance databases for training sample points from which to build our models, thereby expanding the scope of our usability beyond just those applications that we study ourselves.

#### ACKNOWLEDGEMENTS

Part of this work was performed under the auspices of the U.S. Department of Energy by University of California Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48 (LLNL Document Number UCRL-CONF-212365) and under National Science Foundation award CCF-0444413. Any opinions, findings, and conclusions or recommendations expressed in this publication are those of the authors and do not necessarily reflect the views of the National Science Foundation, the Lawrence Livermore National Laboratory, or the Department of Energy.

#### REFERENCES

1. The ASCI purple benchmarkCodes.  
[http://www.llnl.gov/asci/purple/benchmarks/limited/code\\_list.html](http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html) [October 2002].
2. Almsi, G., Archer, C., Castaos, J., Gunnels, J., Erway, C., Heidelberger, P., Martorell, X., Moreira, J., Pinnow, K., Ratterman, J., Steinmacher-Burow, B., Gropp, W., and Toonen, B. Design and Implementation of Message-Passing Services for the Blue Gene/L Supercomputer. *IBM Journal of Research and Devevelopment* 2005. **49**(2/3):393-406.
3. Breiman, L. Bagging predictors. *Machine Learning* 1996. **24**(2):123-140.
4. Brown, P., Falgout, R., Jones, J. Semicarsening multigrid on distributed memory machines. *SIAM Journal on Scientific Computing* 2000; **21**:1823-1834.
5. Carrington, L., Snavely, A., Gao, X., Wolter, N. A performance prediction framework for scientific applications. *Proceedings International Conference on Computational Science Workshop on Performance Modeling and Analysis (PMA03)*, June 2003. Springer LNCS **2659**: Berlin/Heidelberg, 2003; 926-935.
6. Carrington, L., Wolter, N., Snavely, A., Lee, C. Applying an automatic framework to produce accurate blind performance predictions of full-scale hpc applications. *Department of Defense Users Group Conference*, June 2004.
7. Caruana, R., Lawrence, S., Giles, C. Overfitting in neural nets: backpropagation, conjugate gradient, and early stopping. *Proceedings Neural Information Processing Systems (NIPS)*, November 2000. MIT Press: Cambridge, MA, 2000; 402-408.
8. Falgout, R., Yang, U. hypre: a Library of High Performance Preconditioners. *Proceedings of the International Conference on Computational Science (ICCS), Part III*, April 2002. Springer LNCS **2331**: Berlin/Heidelberg, 2002; 632-641.
9. İpek, E., de Supinski, B., Schulz, M., McKee, S. An approach to performance prediction for parallel applications. *Proceedings Euro-Par*, August 2005. Springer LNCS **3648**: Berlin/Heidelberg, 2005; 196-205.



10. İpek, E., McKee, S., de Supinski, B., Schulz, M., Caruana, R. Efficiently exploring architectural design spaces via predictive modeling. *Proceedings Architectural Support for Programming Languages and Operating Systems (ASPLOS XII)*, October 2006. ACM Press: New York, NY.
11. Karkhanis, T., Smith, J. A first-order superscalar processor model. *Proceedings 31st Annual International Symposium on Computer Architecture*, June 2004. IEEE Computer Society Press: Los Alamitos, CA, 2004; 338-349.
12. Kerbyson, D., Alme, H., Hoisie, A., Petrini, F., Wasserman, H., Gittings, M. Predictive performance and scalability modeling of a large-scale application. *Proceedings IEEE/ACM Supercomputing*, November 2001. IEEE Computer Society Press: Los Alamitos, CA, 2001.
13. Lawrence Livermore National Laboratory The ASCI Purple Benchmark Codes. [http://www.llnl.gov/asci/purple/benchmarks/limited/code\\_list.html](http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html), October 2002.
14. Marin G., Mellor-Crummey, J. Cross-architecture performance predictions for scientific applications using parameterized models. *Proceedings International Conference on Measurement and Modeling of Computer Systems (Sigmetrics)*, June 2004. ACM Press: New York, NY; pages 2-13,
15. Mitchell, T. *Machine Learning*. WCB/McGraw Hill, 1997.
16. Petitet, A., Whaley, R., Dongarra J., Cleary, A. HPL—A portable implementation of the high-performance linpack benchmark for distributed-memory computers. <http://www.netlib.org/benchmark/hpl/> [March 2006].
17. Riedmiller, M., Braun, H. A Direct Adaptive Method for Faster Backpropagation Learning: The RPROP algorithm. *Proceedings IEEE International Conference on Neural Networks*, May 1993. IEEE Computer Society Press: Los Alamitos, CA, 1993; 586-591.
18. Saar-Tsechansky, M., Provost, F. Active learning for class probability estimation and ranking. *Proceedings International Joint Conference on Artificial Intelligence (IJCAI)*, August 2001. AAAI Press: Cambridge MA, 2001; 911-920.
19. Yang, T., Ma, X., Mueller, F. Cross-platform performance prediction of parallel applications using partial execution. *Proceedings IEEE/ACM Supercomputing*, November 2005. IEEE Computer Society Press: Los Alamitos, CA, 2005.