PREDICTING PARALLEL APPLICATION PERFORMANCE VIA MACHINE LEARNING APPROACHES

A Thesis

Presented to the Faculty of the Graduate School

of Cornell University

in Partial Fulfillment of the Requirements for the Degree of

Master of Science

by

Karan Singh

August 2007

ABSTRACT

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 thesis, we focus on two well known parallel applications whose variations in execution times are not well understood: SMG2000, 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.

BIOGRAPHICAL SKETCH

Karan Singh received a B.S. in Computer Engineering and a B.S. in Electrical Engineering from Louisiana State University in May 2005. He has been an M.S./Ph.D. student at Cornell's Computer Systems Laboratory since June 2005.

To my parents, Paramjit Singh and Dr. Rajinder Kaur.

ACKNOWLEDGMENTS

I would first like to express my gratitude towards my advisor, Dr. Sally McKee. Her leadership, support, attention to detail, hard work, and scholarship set an example I hope to match some day. I thank Bronis de Supinski and Martin Schulz for their participation and support in this project. I thank Engin Ìpek for his initial work in this area, and the rest of the Fusion group (Pete, Brian, Vince, Chris, Cat, and, of course, Major) for their feedback and commaraderie. Finally, I thank my family and friends. This thesis would not have been possible without their constant support and faith in me.

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 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.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

Chapter 1

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 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 about both the target system and the input space, often leading to loss in accuracy and generality, and failure to capture subtle interactions between architecture and software.

Instead, we 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. Chapter 3 explains network training in detail.

In this thesis, we demonstrate how we construct and use our neural network models for two well known numerical benchmark codes, SMG2000 [6] from the ASCI Purple

benchmark suite [14] and the High Performance LINPACK (HPL) [17] 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 [8], and our techniques are particularly well suited to mining performance databases or to extending fast, parameter-specific models.

Chapter 2

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. We discuss these approaches in Section 2.1, but find it difficult to do an apples to apples comparison because there are few such models freely available to the research community. Proprietary models exist, but are not available for comparisons within the context of the work presented here. In Section 2.2, we compare our approach to a similar sample-based approach using regression.

2.1 Models for Parallel Applications

Marin and Mellor-Crummey [15] 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. 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. [3] 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 [4]. 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. [13] 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. [20] 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 [14]; 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 models less accurate.

2.2 Neural Networks and Regression

In joint work with Lee et al., we compare our neural network approach with piecewise polynomial regression [10]. We construct neural network and piecewise polynomial regression models to prediction application performance for SMG2000 and HPL as a function of the application input parameters. Both approaches predict with similar median and outlier error rates ranging from 2.2% to 10.5%. Given the similarities in prediction results for applications studied, the interesting differences between the approaches lie in the trade-offs that come with using them. Neural networks offer an automated approach and do not require the form of the target function to be known. However, they tend to be treated as a black-box approach because of the difficulty in mapping the learned network as a function of the applied inputs. On the other hand, regression requires statistical techniques such as clustering, association, and correlation analyses to identify relevant predictors and interactions. In addition, one needs to input domain-specific knowledge to guide model construction. Finally, there may be need for additional statistical tests after model construction to ensure model fit and a lack of systematic bias. This requires a background in statistics and work on the part of the modeler since the required steps cannot be automated. Consequently, this may offer a better understanding of the parameter space due to the heavy analysis done beforehand.

Chapter 3

Neural Networks

Machine learning studies algorithms that *learn* automatically through experience. For our problem, we focus on a particular class of machine learning algorithms called *artificial neural networks* (ANNs). They have been used for microarchitectural design space exploration [8], workload characterization [21], and compiler optimization [22]. ANNs automatically learn to predict one or more targets for a given set of inputs. We choose ANNs because they are flexible and well suited for generalized nonlinear regression, and their representational power is rich enough to express complex interactions between variables: any function can be approximated to arbitrary precision by a three-layer artificial neural network [16]. They require no knowledge of the target function, take real or discrete inputs and outputs, and deal well with noisy data.

3.1 Theory

A neural network consists of layers of *neurons*, or switching units: typically, an input layer, one or more hidden layers, and an output layer. Input values are presented at the input layer and predictions are obtained from the output layer. Figure 3.1 shows an example of a fully connected feed-forward neural network. Every unit in each layer is connected to all units in the next layer by weighted edges. Each unit applies an *activation function* to the weighted sum of its inputs and passes the result to the next layer. One can use any nonlinear, monotonic, and differentiable activation function. We use the sigmoid activation function for our models (Figure 3.2 [16]).

Training the network involves tuning edge weights via backpropagation, using gradient descent to minimize error between predicted and actual results. In this iterative

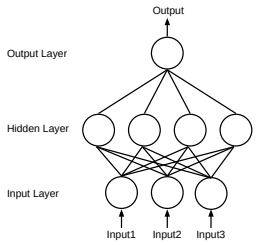


Figure 3.1: Simplified diagram of a fully connected, feed-forward ANN

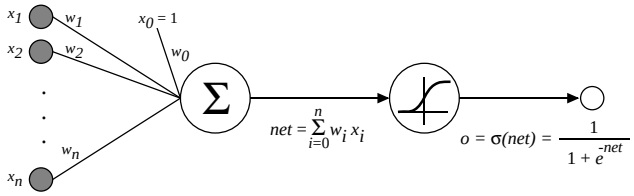


Figure 3.2: Example of a hidden unit with a sigmoid activation function

process, the training samples are repeatedly presented at the input layer, and the error is calculated between the prediction and the actual target. The weights are initialized near zero and are updated using an update rule (similar to the one shown in Equation 3.1) in the direction of steepest decrease in error. As weights grow, the network becomes increasingly nonlinear.

$$w_{i,j} \leftarrow w_{i,j} - \eta \frac{\partial E}{\partial w_{i,j}} \tag{3.1}$$

We use a three-layer neural network with a single hidden layer of 16 units, initial weights drawn uniformly at random from [-0.01,+0.01], and the Rprop learning algorithm, a variant of standard backpropagation. 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 [19].

Neural network models have a tendency to *overfit* on training data, leading to models that generalize poorly to new data despite their high accuracy on the training data. This

is countered by using *early stopping* [5], where we keep aside a validation set from the training data and halt training as accuracy begins to decrease on this set. However, this means we lose some of our training data to the validation set. To address this, we use an ensemble method called *cross validation* to help improve accuracy and mitigate the risk of overfitting the neural network models. This technique consists of splitting the training set into n equal-sized *folds*. Taking n=10, for example, we use folds 1-8 for training, fold 9 for early stopping to avoid overfitting, and fold 10 to estimate performance of the trained model. We train a second model on folds 2-9, use fold 10 for early stopping, and estimate performance on fold 1, and so on. This generates 10 neural networks, and we average their outputs for the final prediction. Each neural network in the ensemble sees a subset of training data, but the group as a whole tends to perform better than a single network because all data has been used to train portions of it. Cross validation reduces error variance and improves accuracy at the expense of training multiple models.

We apply *stratification* to reduce percentage error by replicating each point in the dataset by a factor proportional to the inverse of its target value. As a consequence, the network is trained more on points with small target values, which have large relative but low absolute errors. The absolute error is reduced for these samples to the point that the relative errors across the whole model no longer show large divergences. Prior work [7] finds that stratification significantly improves the performance of backpropagation.

3.2 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 [11].

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.

For this work, we present results for cross validation combined with stratification since it provides the most robust performance, delivering comparable results to other neural network optimization techniques at large training set sizes, and outperforming them at smaller training set sizes [9].

Chapter 4

Experiments

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 SMG2000 and HPL, 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 significantly different architectures. This enables predicting runtime or other important characteristics across large input parameter spaces with high dimensionality.

4.1 Target Applications and Their Characteristics

We study two well known numerical applications: SMG2000, a semicoarsening multigrid solver based on the *hypre* library [6]; and HPL, a portable implementation of High Performance LINPACK [17] used to solve (random) dense linear systems on distributed memory computers.

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 4.1. 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, which makes predicting performance difficult. Consequently, only a rough analytical model approximating the communication volume for cubic working set sizes has been built to date [2]. While it would be possible to extend this model for arbitrarily shaped working sets, doing so would be extremely complex, and the result would likely be in-

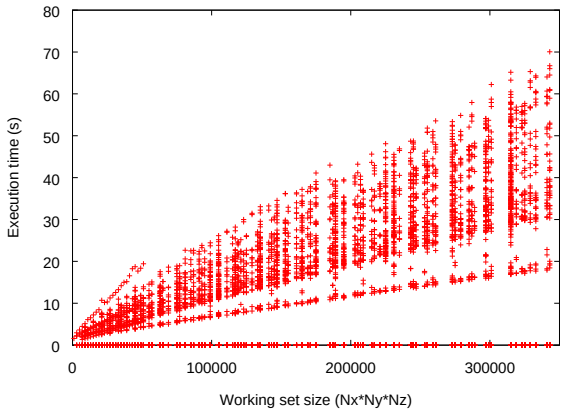


Figure 4.1: Execution times for SMG2000 for varying processor workloads (N_x,N_y,N_z) and processor topologies (P_x,P_y,P_z) running on 512 Blue Gene/L nodes

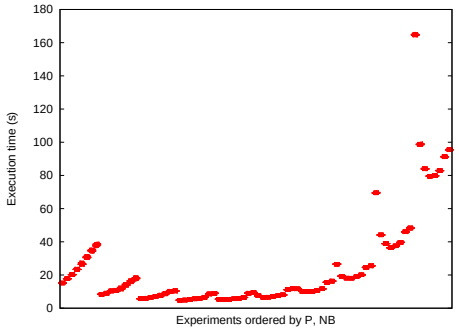


Figure 4.2: Execution times for HPL with varying block sizes (NB) and topologies (P) running on 512 Blue Gene/L nodes

tractable. 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 many parameters. However, only rough guidelines exist on how to choose the best settings for a given target architecture [17], forcing the user to rely on hand-tuning for each platform. We choose five parameters that are generally known to be most significant [18], and we vary processor topology in tandem. Again, we observe that application execution times change dramatically with varying parameters, as shown in Figure 4.2.

4.2 Results

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 4.1 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 [1].

Table 4.1: Platform parameters

	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 4.2: 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 describe the processor topology in all three dimensions; the problem size for a particular run is a volume of size $N_x * P_y \times N_y * P_y \times N_z * P_z$.

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 4.3: 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 [17])

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

For both SMG2000 and HPL we explore a six-dimensional parameter space. Table 4.2 and Table 4.3 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 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 4.4 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

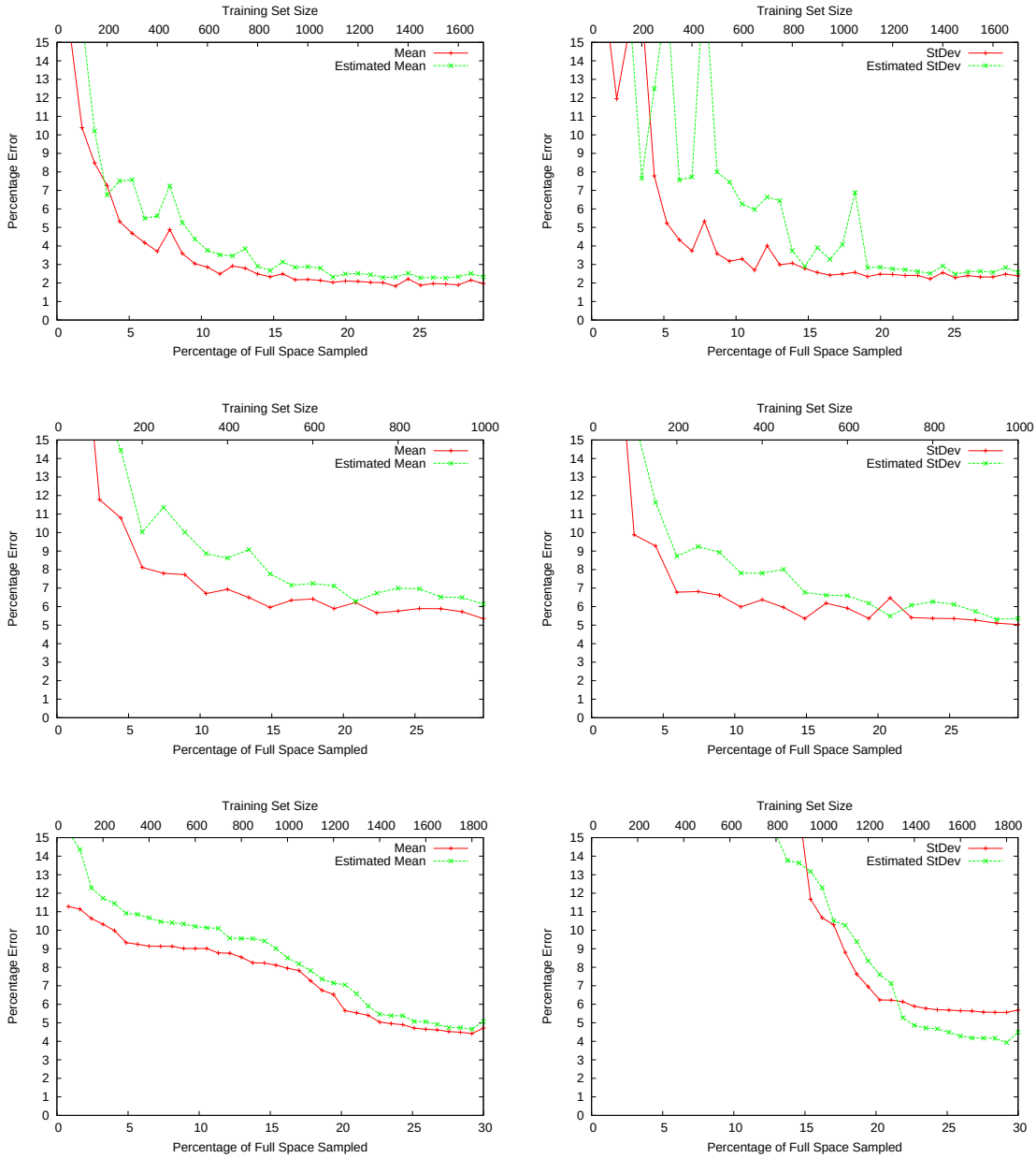
Table 4.4: Runtime statistics for each dataset

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

representation of the full parameter space, but our models can interpolate to predict performance for intermediate values. We iteratively train and test our models, incrementing the sample dataset by 50 points at each round. We randomly select a test set of 1K points from remaining data and report model accuracy on this set.

Figure 4.3 shows learning curves for mean prediction error (left column), and standard deviations for those prediction errors (right column). All graphs show results when we use Rprop with 10-fold cross validation for training, and stratification to select sample points. The top graphs show results for HPL on Blue Gene/L, and the middle and bottom graphs show results for SMG2000 on Blue Gene/L and Thunder respectively. Predicted values for mean error and standard deviation are derived from the multiple models in our ensemble method, and actual values show comparisons of model predictions against the full dataset.

Table 4.5 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, the average prediction error on the test set and corresponding standard deviation are 9.24% and 38.37%. 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



(a) prediction error mean (b) prediction error standard deviation

Figure 4.3: Results for HPL on Blue Gene/L (top), SMG2000 on Blue Gene/L (middle), and SMG2000 on Thunder (bottom)

Table 4.5: Mean errors and standard deviations as training set size increases

	training set size	error	stdev
SMG2000	5%	9.24	38.37
on	10%	9.01	27.58
Thunder	20%	5.66	6.23
SMG2000	5%	8.12	6.78
on	10%	6.70	5.99
BG/L	20%	6.22	6.47
HPL	5%	4.68	5.23
on	10%	2.86	3.31
BGL	20%	2.09	2.46

increases, error decreases, showing that the model benefits from the additional information included in the dataset at each round of training. 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. The HPL dataset is the most predictable: our models predict performance with only 2-3% error and similar standard deviations.

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 20% of our datasets, our models achieve accuracies of about 93-94% on SMG2000 and 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 1-10 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.

Chapter 5

Conclusions

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 spaces. We 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. We develop application-specific performance models to enable predicting runtime or other important characteristics across large input parameter spaces with high dimensionality. 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. Improving our ability to configure these kinds of applications to run most effectively enables the faster solution of much larger problems in scientific computing. 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.

BIBLIOGRAPHY

[1] 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.

[2] Brown, P., Falgout, R., Jones, J. Semicoarsening Multigrid on Distributed Memory Machines. *SIAM Journal on Scientific Computing* 2000; **21**:1823-1834.

[3] 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.

[4] 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.

[5] 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.

[6] 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.

[7] Ìpek, E., de Supinski, B.R., Schulz, M., McKee, S.A. An Approach to Performance Prediction for Parallel Applications. *Proceedings Euro-Par*, August 2005. Springer LNCS **3648**: Berlin/Heidelberg, 2005; 196-205.

[8] Ìpek, E., McKee, S.A., Singh, K., de Supinski, B.R., Schulz, M., Caruana, R. Efficient Architectural Design Space Exploration via Predictive Modeling. *ACM Transactions on Architecture and Code Optimization*, to appear, 2007.

[9] Singh, K., Ìpek, E., McKee, S.A., de Supinski, B.R., Schulz, M., Caruana, R. Predicting Parallel Application Performance via Machine Learning Approaches. *Concurrency and Computation: Practice and Experience*, May 2007. *http://dx.doi.org/10.1002/cpe.1171*

[10] Lee, B., Brooks, D., de Supinski, B.R., Schulz, M., Singh, K., McKee, S.A. Methods of Inference and Learning for Performance Modeling of Parallel Applications. *Proc. ACM Symposium on the Principles and Practice of Parallel Programming*, March 2007.

[11] Singh, K., McKee, S.A., de Supinski, B.R., Schulz, M. Using Machine Learning to Explore Huge Parameter Spaces for High End Computing Applications: Tools and Examples. *Cornell Computer Systems Lab Technical Report CSL-TR-2007-1049*, July 2007.

[12] 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.

[13] 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.

[14] Lawrence Livermore National Laboratory The ASCI Purple Benchmark Codes. *http://www.llnl.gov/asci/purple/benchmarks/limited/code_list.html*, October 2002.

[15] 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,

[16] Mitchell, T. *Machine Learning.* WCB/McGraw Hill, 1997.

[17] 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].

[18] de Supinski, B.R. *Private Communication*, March 2006.

[19] 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.

[20] 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.

[21] Yoo, R., Lee, H., Chow, K., Lee, H. Constructing a Non-Linear Model with Neural Networks for Workload Characterization. *IEEE International Symposium on Workload Characterization (IISWC)*, San Jose, CA, October 2006; 150-159.

[22] Dubach, C., Cavazos, J., Franke, B., Fursin, G., O'Boyle, M., Temam, O. Fast Compiler Optimisation Evaluation Using Code-Feature Based Performance Prediction. *CF '07: Proceedings of the 4th International Conference on Computing Frontiers*. New York, NY, USA: ACM Press, 2007; 131-142.