An Approach to Performance Prediction for Parallel Applications

Engin Ipek[1], Bronis R. de Supinski[2], Martin Schulz[2], and Sally A. McKee[1]

[1] Computer Systems Lab
School of Electrical and Computer Engineering
Cornell University
{engin,sam}@csl.cornell.edu
[2] Center for Applied Scientific Computing
Lawrence Livermore National Laboratory
Livermore, CA 94551
{bronis,schulzm}@llnl.gov

Abstract. Accurately modeling and predicting performance for large-scale applications becomes increasingly difficult as system complexity scales dramatically. Analytic predictive models are useful, but are difficult to construct, usually limited in scope, and often fail to capture subtle interactions between architecture and software. In contrast, we employ multilayer neural networks trained on input data from executions on the target platform. This approach is useful for predicting many aspects of performance, and it captures full system complexity. Our models are developed automatically from the training input set, avoiding the difficult and potentially error-prone process required to develop analytic models. This study focuses on the high-performance, parallel application SMG2000, a much studied code whose variations in execution times are still not well understood. Our model predicts performance on two large-scale parallel platforms within 5%-7% error across a large, multi-dimensional parameter space.

1 Introduction

With rising architecture and software complexity, it becomes increasingly difficult to accurately model and predict performance for large-scale applications. Analytic models often fail to capture subtle interactions between architecture and software. Furthermore, they usually must be constructed manually in a long and often error-prone process. In this paper, we address these problems with the help of machine learning techniques. We gather performance samples from multiple executions of an application, and use this data to *automatically* construct performance models by training multilayer neural networks. Since we take input data from executions on the target platform, we capture full system complexity, without having to manually model architectural details. Our approach is useful for a wide range of application performance prediction problems. Our techniques are particularly well suited to mining performance databases or to extend fast, parameter-specific models.

J.C. Cunha and P.D. Medeiros (Eds.): Euro-Par 2005, LNCS 3648, pp. 196–205, 2005.

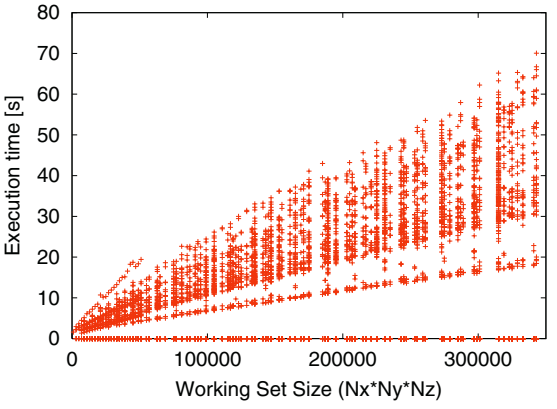

Fig. 1. Execution times for SMG2000 for varying processor workloads (Nx,Ny,Nz) and processor topologies (Px,Py,Pz) running on 512 nodes on BlueGene/L

Here we focus on SMG2000, a semicoarsening multigrid solver based on the *hypre* library [4]. We develop application-specific performance models for parallel architectures, enabling prediction of runtime or other important characteristics across a large input parameter space with high dimension. SMG's six-dimensional parameter space describes both shape of the workload per processor and 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 5×. Although SMG has been studied extensively and an analytic model describing communication requirements exists [1], the code's variations in execution time are not well understood (partly due to SMG's complex, recursive algorithm). The analytic model is restricted to cubic workloads and only describes communication complexity; it is not designed to represent architectural details. Extending it for arbitrarily shaped workloads is possible, but would be extremely complex, and the result would likely be intractable. Worse, adding architectural features is infeasible. Our automatic, empirical modeling approach overcomes these limitations *without knowledge of the application or algorithms.*

We demonstrate how we use neural networks to construct our models, and we identify the two major challenges of this approach: avoiding noise in the dataset, and choosing an appropriate sampling technique for the training phase of the neural network. The latter is necessary to avoid a bias toward short runtimes, since those exhibit a higher relative error. To correct this skew, we develop new functions that scale error by the runtime of the training samples. The resulting model can predict SMG2000's performance on two large-scale parallel platforms within 5%-7% error across a large, multi-dimensional parameter space.

2 Approach

We use machine learning models to predict application performance across a large, multidimensional parameter space defined by program inputs. We first

collect a sample dataset by choosing a collection of points spread regularly across the parameter space; we obtain performance results for these on actual hardware. We reserve a portion of this dataset as a *test* to report the final performance of our models, and never train on this data. Next, we randomly separate the remainder of the data into *training* and *validation* sets, where the former is used to adjust model parameters through a learning algorithm, and the latter is used to assess the performance 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 not included in our training or validation sets.

2.1 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. Figure 2 shows an example neural network architecture. The network is composed of a set of *units* that process the value at their inputs and produce a single scalar value. These values are then multiplied by a set of weights and communicated to other units within the network. Each edge in Figure 2 represents a weight, and each node represents a unit. The set of incoming edges at each unit indicates the set of values communicated to it. In this specific network architecture, the input parameters are placed at the first (lowest) layer, and information flows from bottom to top. The units that produce the final predictions are *output units*, and those that receive input parameters are *input units* (input units simply pass incoming values to all of their outgoing edges). In addition, one or more layers of *hidden units* may be part of the network architecture. Hidden units process the outputs of other units, and, in turn 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. Every unit in a given layer receives values from all units in the layer below it, and hence this type of ANN architecture is called a *multilayer fully connected feedforward* neural network. Figure 2 shows a feedforward neural network with three input units, one output unit, and a single layer of four hidden units.

At each step during training, a new example is presented at the network's input layer. At each layer, every unit forms a weighted sum of the incoming values and associated weights. This sum is then processed by an *activation function* that produces the output of that unit. In this study, we use fully connected feedforward neural networks with the sigmoid function as the activation function. Figure 3 shows the operation of the sigmoid activation function on the weighted sum of inputs (depicted immediately right of the summation in Figure 3) to form the unit's result output. After a prediction on the current example, the weights in the network are updated in proportion to their contribution to the error.

Other types of predictive models may be applied to performance (see Section 4). Here we limit our scope to ANNs for three reasons. First, ANNs permit target values and inputs/outputs to be discrete, continuous, or a mix, allowing them to perform well in both regression and classification problems and to learn

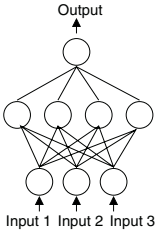

Fig. 2. A feedforward neural network with a single hidden layer

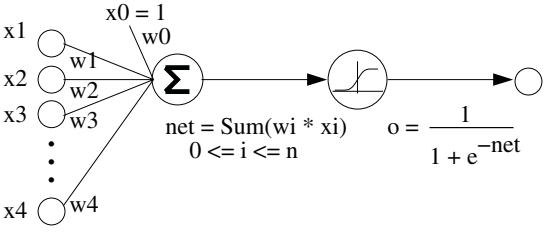

Fig. 3. A network unit with sigmoid threshold activation function (reproduced from Mitchell [8])

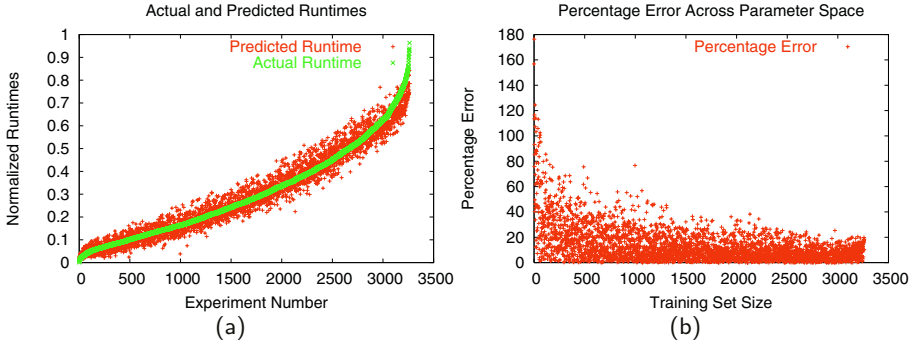

Fig. 4. Comparison of (a) predicted vs. actual performance and (b) percentage error

from various types of attributes describing performance prediction problems. Second, ANNs need not know the form of the target function in advance. Third, ANNs tend to work well with possibly noisy data, making them ideal for training on performance results collected in the presence of system noise.

2.2 Application to Performance Prediction

Figure 4 shows results of an initial performance prediction study consisting of 13.2K data points. A standard, fully connected feedforward neural network with 16 hidden units is trained on 10K points, and predictions are made on the re-

maining 3.2K. Despite training on a large portion of the parameter space, in most cases model accuracy is low. Average test error is 13.8% with a standard deviation of 14.8%—excessively high for performance prediction purposes.

Poor accuracy of standard feedforward neural networks on this dataset results from two factors. 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. This imposes a fundamental limit on model accuracy for future datasets. Second, the training algorithm that adjusts network weights is unsuitable for reducing percentage error. By default, the backpropagation training algorithm tries to reduce absolute mean-squared-error. During training, examples on which the model makes higher absolute error are given greater weight, even though this error may be small in relative terms as a percentage of the target value. Given two test cases t_1 and t_2, where runtime of t_1 is 100 seconds and of t_2 is 1 second, an error of 0.5 seconds is given equal weight for both, even though the percentage error varies drastically between the two examples (0.5% vs. 50%).

2.3 Required Network Refinements

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, BlueGene/L), this problem is either nonexistent or negligible. On other platforms, we find that reserving at least one processor per node for system processing greatly alleviates noise.

Once noise levels are acceptably diminished, a mechanism for training the neural network to reduce percentage error is needed. We combine a sampling technique called *stratification*, and an ensemble learning mechanism called *bagging* (bootstrap aggregation). Stratification replicates each point in the dataset by a factor proportional to the inverse of its target value such that, during training, the network sees points with small target values many more times than it sees those with large absolute values. As a result, the training algorithm puts varying amounts of emphasis on different regions of the search space, making the right tradeoffs when setting weights to minimize percentage error. We apply bagging to train an ensemble of models from the dataset, averaging predictions from the ensemble to reduce model variance.

3 Experiments

We present results of applying our technique to performance prediction of SMG on the Thunder and BlueGene/L systems at Lawrence Livermore National Laboratory. Architectural features of these systems on which data is taken are detailed in Table 1. Table 2 shows program parameters. For the BlueGene/L dataset, we

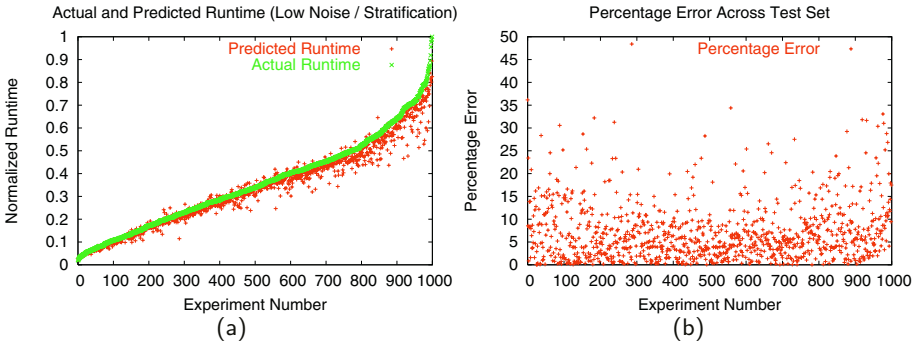

Fig. 5. Comparison of (a) predicted and actual performance and (b) percentage error

Table 1. Platform parameters

	BlueGene/L	Thunder
Processor	IBM BlueGene	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	8GB DDR266
Network	3D Torus + Global Combine/Broadcast Tree Network	Fat Tree (Quadrics QsNet)
Processors Used/Node	1/2	3/4
Number of Nodes Used	512	64

Table 2. Application parameters

Parameter	BlueGene/L	Thunder
Nx	10-510 in steps of 20	10-250 in steps of 30
Ny	10-510 in steps of 20	10-250 in steps of 30
Nz	10-510 in steps of 20	10-250 in steps of 30
Px	1,8,64,512	1,3,4,12,16,48,64,192
Py	1,8,64,512	1,3,4,12,16,48,64,192
Pz	1,8,64,512	1,3,4,12,16,48,64,192
Px*Py*Pz	512	192
Nx*Ny*Nz	1000>Nx*Ny*Nz>343000	216000>Nx*Ny*Nz>9261000

keep 1K random samples for final testing only (we do not train on these points) and report the accuracy of our model on this data. Similarly, we separate 1.3K data points for testing in the Thunder dataset.

Figure 6(a) shows a learning curve that indicates how the accuracy of the neural network changes as the size of the training set is increased for the BlueGene/L dataset. At a training set size of 250 points, the average error on the

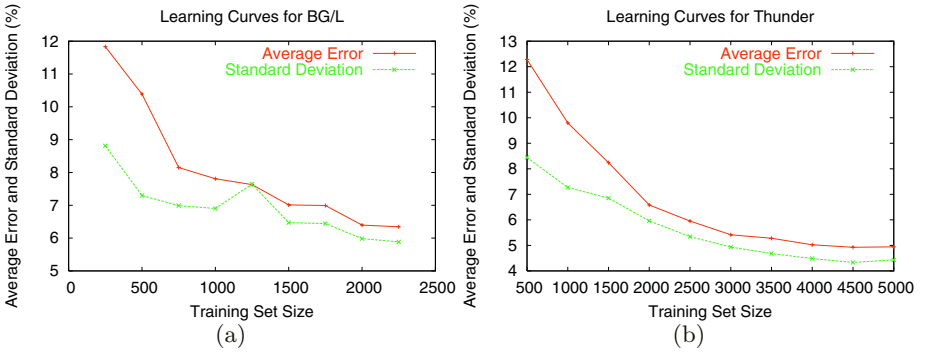

Fig. 6. Learning curves showing how average error and standard deviation improve with training set size for (a) BlueGene/L and (b) Thunder

test set is nearly 12.3%, and the standard deviation of error across the test set is 8.7%. At this point, the training set is too small and contains too little information to build a highly accurate model. As training set size increases, error decreases sharply, showing that the model benefits significantly from the additional information included in the dataset at each point. Eventually, the curves begin to flatten, as any additional data presented to the network contains only incremental new information. When 2.25K of the 3.25K total points are used for training, the error rate of the of the network falls to 6.7%. Similarly, the standard deviation of the error decreases with increasing training set size.

Thunder's learning curves (Figure 6(b)) follow the same trends. With 500 data points, the error rate on the test set is 12.28%. The error falls sharply as more data points are added, reaching 5.4% at a training set size of 3K. Further increases in training set size result in diminishing improvements, and at a training set size of 5K points, the network achieves 4.9% error. Similarly, the standard deviation ranges from 8.4%-4.4% between 500-5K points.

The results indicate that the accuracy of our approach can be quite high given enough training points. The size of the parameter space is much, much larger than the total number of points we have collected. We *sparsely* step through the SMG2000 parameters to obtain our dataset. Therefore, our approach is easily applicable to learning from performance databases that contain results for a sparse sampling of parameters. In addition,the amount of time required to train a model ranges between 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 much more efficiently than most analytical models.

4 Related Work

Other approaches to performance prediction include analytic models. Space prevents our providing a full treatment of related work, but Karkhanis and Smith [5] give an excellent review of prior work in architectural performance prediction.

Marin and Mellor-Crummey [7] semi-automatically measure and model program characteristics, predicting application behavior based on properties of the architecture, properties of the binary, and application inputs. Their toolkit provides a set of predefined functions, and the user may add customized functions to this library if the set of existing functions is too restrictive. In contrast to our work, they vary the input size in only one dimension, and they cannot account for some important architectural parameters, such as cache associativity in their memory reuse modeling. Our six-dimensional space would make use of their approach much more difficult, significantly increasing the number of required samples as well as the search space for the best analytic function (as a weighted sum of given base functions along each parameter dimension).

Carrington et al. [2] develop a framework for predicting performance of scientific applications, demonstrating its effectiveness on LINPACK and an ocean modeling application. The approach is built on a convolution method that represents a computational mapping of an application signature onto a machine profile. Simple benchmark probes create the machine profiles, and a separate tool generates the application signatures. Extending the convolution method allows them to go from modeling kernels to whole benchmarks to full-scale HPC applications [3]. This automated approach relies on the generation of several traces, delivering predictions with accuracies of between 4.6 and 8.4%, depending on the sampling rates of those traces. Using full traces obviously gives the best performance, 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 accuracies range from 0.1 to 8.7% on different platforms. This work is complementary to our own, and the two approaches may work well in combination. The analytic models could provide the bootstrap data, and our models could give them full application input parameter generality.

Kerbyson et al. [6] present a highly accurate, predictive analytical model that encompasses the performance and scaling characteristics of SAGE, a multidimensional hydrodynamics code with adaptive mesh refinement. As with the model presented here, 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. They validate the 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. Their application-centric modeling approach requires static analysis of the code: a detailed model must be developed for each application of interest.

Karkhanis and Smith [5] construct a first-order model of superscalar microprocessors. Their approach is intuitive, provides insight, and is reasonably accurate, finding that their performance estimates are between five and 13% accurate with respect to detailed simulations of the applications they study. The model's analytic core incorporates cache and branch predictor statistics gathered from functional-level trace driven simulation. They target uniprocessors,

and while intuitive, the approach is largely ad hoc and currently limited in the architectural features it models. Their model is more appropriate for studying proposed architectures, whereas we predict performance on existing platforms.

5 Conclusions and Future Work

We have presented a machine learning approach to application performance prediction—multilayer neural networks—and have refined and adapted this approach to yield highly accurate results for SMG2000 on two different high-performance platforms. Our approach is especially attractive for its ease of use and its obliviousness to details of application internals. This makes it ideal for mining performance databases to make performance predictions. While promising, this approach still presents some challenges in making it generally useful in the absence of an existing database. The time required to gather each data point in the training set is larger than we would like, for instance. Reducing the number of points required in our training datasets is one promising direction of current research.

Acknowledgments

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 by by the National Science Foundation under award ST-HEC 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. The authors thank Rich Caruana and the anonymous referees for their valuable feedback on this work.

References

1. P. Brown, R.D. Falgout, and J.E. Jones. Semicoarsening multigrid on distributed memory machines. *SIAM J. Sci. Computing*, 21:1823–1834, 2000.
2. L. Carrington, A. Snavely, X.Gao, and N. Wolter. A performance prediction framework for scientific applications. In *International Conference on Computational Science Workshop on Performance Modeling and Analysis (PMA03)*, pages 926–935, June 2003.
3. L. Carrington, N. Wolter, A. Snavely, and C.B. Lee. Applying an automatic framework to produce accurate blind performance predictions of full-scale HPC applications. In *Department of Defense Users Group Conference*, June 2004.
4. R.D. Falgout and U.M. Yang. hypre: a Library of High Performance Preconditioners. In *Proceedings of the International Conference on Computational Science (ICCS), Part III, LNCS vol. 2331*, pages 632–641, April 2002.

5. T.S. Karkhanis and J.E. Smith. A first-order superscalar processor model. In *Proceedings of the 31st Annual International Symposium on Computer Architecture*, pages 338–349, June 2004.
6. D.J. Kerbyson, H.J. Alme, A. Hoisie, F. Petrini, A.J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of IEEE/ACM Supercomputing '01*, November 2001.
7. G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems (Sigmetrics '04)*, pages 2–13, June 2004.
8. T.M. Mitchell. *Machine Learning*. WCB/McGraw Hill, Boston, MA, 1997.