Simulating Large Scale Parallel Applications using Statistical Models for Sequential Execution Blocks

Gengbin Zheng, Gagan Gupta, Eric Bohm, Isaac Dooley, and Laxmikant V. Kalé
Department of Computer Science
University of Illinois at Urbana-Champaign
Urbana, IL 61801, USA
Email: {gzheng, gagan, ebohm, idooley2, kale}@illinois.edu

***Abstract*—**

Predicting sequential execution blocks of a large scale parallel application is an essential part of accurate prediction of the overall performance of the application. When simulating a future machine that is not yet fabricated, or a prototype system only available at a small scale, it becomes a significant challenge. Using hardware simulators may not be feasible due to excessively slowed down execution times and insufficient resources. These challenging issues become increasingly difficult in proportion to scale of the simulation.

In this paper, we propose an approach based on statistical models to accurately predict the performance of the sequential execution blocks that comprise a parallel application. We deployed these techniques in a trace-driven simulation framework to capture both the detailed behavior of the application as well as the overall predicted performance. The technique is validated using both synthetic benchmarks and the NAMD application.

***Index Terms*—parallel simulator, performance prediction, trace-driven, machine learning, statistical model**

I. Introduction

Emerging applications such as computational biology, computational cosmology, engineering design, earthquake modeling and weather forecasting demand an unprecedented amount of compute power. On the other hand, computer hardware designs have achieved incredible performance gains over the past half century, and the trend toward more powerful and cost-effective systems continues. As an example, the new Blue Waters supercomputer will have more than $300,000$ cores, targeting sustained petaflops of performance for a range of science and engineering applications.

Developing parallel applications that scale up to use these machines efficiently is a significant challenge. Each application evidences idiosyncratic memory usage, communication pattern, and load balance issues. Furthermore, for irregular and dynamic applications, the runtime behaviors of a parallel application can be very complicated. Therefore, parallel performance is hard to model without actually running the program. Large-processor runs on supercomputers are costly and rarely available as often as desired. When the target machine is not operational yet, these challenges become even greater. The prevalent practice is therefore not to begin tuning and scaling until the machine becomes available, creating a large time lag between machine availability and its productive lifetime. A simulator that can simulate the behavior of parallel applications and predict their performance using small clusters thus becomes increasingly important for application developers and machine designers.

To achieve a high fidelity prediction of a parallel application, one needs to accurately predict the performance of CPU in the sequential execution blocks and of the interconnect network. The sequential performance issue is the focus of this paper. Low availability of a sufficiently large target platform for the full scale application often motivates the use of an available platform with different performance characteristics. Accurate prediction under such conditions is only possible with a deep understanding of the differences and a framework that compensates for them accordingly. Even when actual hardware is available, its earliest available form is typically a relatively small scale early prototype system. Simulation for a full system can still be challenging, since the prototype system may not be large enough to run the application with a problem size appropriate for the full machine.

One commonly used tool to predict the sequential execution time of applications for future machines is the cycle accurate hardware simulator. However, using such a simulator may not be feasible due to a significant slowdown (e.g. 10,000X) in execution time. If the target problem would require 1 second on $300,000$ cores, then, assuming perfect scaling, one would have to wait 95 years for those results from a typical single core cycle accurate hardware simulator. Such naive uses are clearly impractical. In contrast, we will show that simulation and modelling can be efficient and accurate.

This paper introduces two techniques to predict performance of sequential execution blocks of large scale applications effectively in a trace-driven simulation system. The first technique is a selective sampling technique, either using *slicing* [4], or *miniaturization* to reduce the required resource footprint of cycle accurate execution. The second uses machine learning algorithms to effectively model the execution times of the sequential execution blocks. These techniques are then validated using the NAS BT benchmark, the synthetic kNeighbor benchmark, and the NAMD [11] application. In particular, we compare a detailed performance prediction of NAMD on an Argonne's BlueGene/P installation against a 4096-core run.

II. BigSim Simulation Framework

We implemented the proposed statistical model-based prediction methods within the context of the BigSim simulation

framework. BigSim is a simulation framework that aims at providing fast and accurate performance evaluation of current and future large parallel systems using much smaller machines, while supporting different levels of fidelity. It targets petascale systems composed of hundreds of thousands of multi-core nodes while enabling researchers to realistically model the performance of a specific interprocessor network design running a specific scientific application code. BigSim is capable of simulating a broad class of applications written in CHARM++ and MPI. It is currently being used to predict the performance of applications on the upcoming Blue Waters system.

BigSim consists of two components. The first component is a parallel *emulator* [18], or functional simulator, that provides a virtualized execution environment that mimics the target machine at full scale using much smaller clusters. It uses techniques such as out-of-core execution [9] to handle applications with large memory footprint. This emulator generates a set of event logs during execution that capture the application's computation and communication behaviors. The second component is a post-mortem trace-driven parallel *simulator* [19] that predicts parallel performance using the event logs as input, and supports multiple resolutions for prediction of sequential and network performance. For example, a high resolution simulation would predict communication performance accurately by simulating each packet of every message flowing through the network NICs and switches, using a parallel discrete event simulation technique. This paper focuses on the issues surrounding accurate performance prediction for sequential pieces of code. However, the network simulator component is applied in the case studies in Section IV.

An important reason for simulating parallel applications is to analyze the simulated behavior of the application. The results from the simulation could reveal both the absolute performance of the application as well as detailed measurements that help elucidate algorithmic or implementation problems within an application or bottlenecks within some particular interconnect design. Furthermore, when combined with a parallel debugger, BigSim provides a realistic virtualized environment for effective debugging [4].

With the trace-driven simulation, rich performance data can be generated by the BigSim simulator in the Projections [6] log format, which can then be analyzed by the numerous tools in the Projections performance visualization toolkit. An example of using Projections will be demonstrated in Section IV.

A. Dependent Execution Blocks

When an application is run within the BigSim emulator, the emulator records a set of dependencies between the *dependent execution blocks* (DEBs) which comprise the execution of the application. Each DEB represents a segment of sequential execution within the program during which a task (or MPI process or CHARM++ *entry method*) does not block or wait for remote data. Each DEB is triggered by the receipt of one or more messages and the completion of zero or more other DEBs. For example, in normal MPI use with blocking receives, each DEB depends on one message and one preceding DEB. When an MPI waitall is used, the subsequent DEB would depend on many messages and one preceding DEB.

The time interval of a DEB is divided into *sequential execution blocks* (SEBs), with the proviso that there is an SEB boundary at each message-spawn. These SEBs can be further divided in case it is easier to trace and predict individual function call times. Typically, SEBs represent computational intensive functions in the program.

The emulator stores DEB data in its messaging logs. Figure 1 illustrates the structure of a DEB and the data stored for each one. Each DEB is assigned a unique global ID. The BigSim emulator records which entity (such as an MPI process, a virtual processor, or a chare) corresponds to each DEB, the time duration of the DEB, a predecessor list (DEBs and messages) and a successor list (DEBs). In addition, it records a set of messages spawned by the DEB along with the time at which they were spawned, specified as an offset in time from the start of the DEB. Each emulating processor produces a log file containing the DEBs for all entities emulated on that processor.

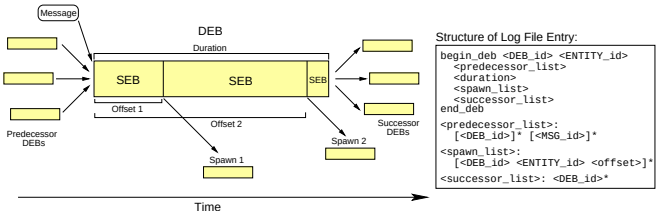


Fig. 1. Dependent Execution Block.

It can be shown that for realistic models of computations (in either MPI/AMPI or CHARM++), this information suffices to reconstruct execution even if messages were to be delivered in a different order [19].

B. Predicting Sequential Execution Blocks

While the communication patterns of the parallel program are recorded as a set of DEBs for modeling network performance, the other important aspect of performance prediction is to accurately model the time spent executing the sequential portions of code, namely the SEBs, that are within each DEB.

When the emulator is run on some specific system, it can record the durations of SEBs as they run on that system. However, these recorded measurements might not be useful when attempting to predict the performance of an entirely different type of system.

Several approaches to predicting SEB performance at different resolutions have previously been explored in BigSim:

1) **User-supplied SEB durations:** Users specify the durations of each SEB as would be expected for a run on the target machine using a call-back API from the emulator. This is a simple and powerful approach, since the API can be used to implement either simple or user defined models. However, it may not be feasible for the user to specify the durations of all SEBs for a large scale application.

2) **Simple scaling:** Wall-clock measurements of the durations of SEBs on the emulating machine can be multiplied by a suitable multiplier (scale factor) to obtain the predicted running time on the target machine. This multiplier might be a function of the clock speeds of the simulating and target machines. The user simply provides the scaling factor.

3) **Cycle-Accurate Models:** When a more accurate prediction is desired, and the target processor is not yet fabricated, one can utilize a cycle-accurate simulator of the target processor to predict the sequential performance of SEBs. The challenge here is compounded by the fact that such simulators are often slower than real processors by a factor of 10,000 or more. Even though many chip simulators (such as IBM's SystemSim simulator [5]) have the capability of fast forwarding with only a 10 fold slowdown (or so) through regions of code that do not require accurate prediction, it is often infeasible to use them to simulate large scale applications.

These approaches either do not provide accurate prediction, or do so at the cost of extremely high overhead. This motivates the work in the paper to find an efficient and accurate prediction method. The resulting schemes described later in the paper approach the accuracy of a cycle accurate simulation of the whole program, without requiring the costly time of actually running the whole program within a cycle accurate simulation.

III. SEB Prediction

The approach proposed in this paper to predict SEB performance is as follows: (A) identifying major SEBs that constitute the majority of the execution time of the program and choose application-specific parameters that best represent each type of SEB; (B) emulating the application at full scale (on any available machine) to generate application traces, (C) executing the application in a small scale (C_1) or only on a few processors (C_2) of a real system, or cycle accurate simulator, to collect data relating the SEB execution times to the associated parameters; (D) building SEB duration models for each SEB based on the obtained data via machine learning techniques; (E) using the models to replace SEB durations in the full scale traces; and finally, (F) running the high resolution simulator to predict the overall application performance. This approach is graphically portrayed in Fig. 2.

Note that the full scale application need only be emulated once. The trace-driven BigSim simulation in the last step does not re-execute any application code, but instead uses the trace logs produced by the emulator. This reduces the need for running the application multiple times due to changes in the network topology or processor architectures being modeled. The rest of this paper describes methods for predicting SEB execution times when parts of an application can be run either on the same type of processor as the target machine or on a cycle-accurate simulator.

A. Identifying Key SEBs

It is important to identify key SEBs (Step A) within an application if more than one type of SEB exists. Furthermore it is helpful to prioritize user attention on SEBs which are likely

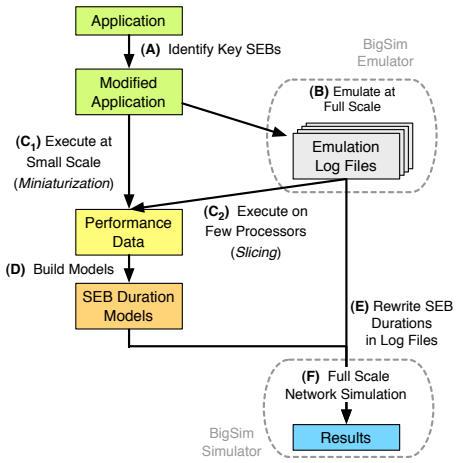


Fig. 2. Overview of the proposed simulation workflow, using the existing BigSim Emulator, BigSim Simulator, and new SEB performance models.

to significantly effect performance. Applications, in some cases, need to be modified in order to allow the emulation system to know which SEB is currently being executed. Thus it is useful for the application developer to identify and add the appropriate markers to the computationally intensive SEBs. The application developer would have such domain knowledge to guide the process, or the SEBs could be determined with the help of profiling tools such as gprof.

In order for the simulation system to learn the computational behavior in these SEBs, a set of key parameters that can affect the performance of an SEB are also provided to the simulation system by the developer, which will be used later to build statistical models for SEBs. For example, SEBs for a particular function (e.g. integration of forces to calculate new particle positions) in NAMD would likely have associated parameters that specify the number of atoms which represents the amount of computation performed in the SEB. Parameters for the SEBs can sometimes be difficult to get, for example, when the code has many *conditional* statements that significantly vary a function's behavior and the amount of computation across different executions of that function. In this case, the execution time on the emulating machine, or some measurement of performance counters are useful for the modelling.

The BigSim framework is flexible enough to allow users to record these application-specific parameters and even performance counters corresponding to each SEB during emulation.

B. Data Collection At Small Scale

Once the application's SEBs and corresponding parameters have been annotated, the application needs to be run on the target system (either real hardware, or hardware simulator), where the SEB native execution time (optionally with performance counters) related to the parameters is measured.

One significant challenge in collecting data in order to build a statistical model for a large parallel program, as mentioned earlier, is the execution of enormous numbers of SEBs on small scale (prototype) machine. This might be infeasible due to memory limitations or be very time consuming. This may also cause changes in the application behavior due to

constraints on memory or network bandwidth. Executing all SEBs within a cycle-accurate simulator, for example, might also be prohibitively expensive. Thus, it is essential to develop techniques for predicting SEBs execution times with high fidelity while only executing a small fraction of them in an accurate, but inexpensive manner.

Several approaches to scale down the number of SEB executions are proposed and described below. These approaches are orthogonal to each other and could be used in combination with each other, if desired.

1) **Slicing:** At Step B (Fig. 2), a full scale application is executed on the emulator, the emulator logs the contents of all incoming messages processed by a user-selected subset of processors (*sliced processors*). This emulation takes place with the highest ratio of emulators to emulating processors which will fit in the memory of the emulating machine and complete in reasonable time. In Step C_2, the emulation logs can now be executed on a prototype machine, or cycle accurate simulator, to replay the execution of any of these independently. Typically one replays as many of the selected processors as is practical within the constraints of the replaying platform. During replaying, the incoming messages which otherwise would have been received from the network are loaded from the previously recorded log files. This slicing scheme, also called *processor extraction* was first introduced in BigSim emulator [4]. A simple variant of this scheme is also implemented for MPI, in which case, all the return values of each MPI call are stored in log files. This approach allows selective replaying of any processor sequentially in the application to record the native performance of SEBs.

Pros: Since the exact sequence of messages is replayed on one chosen target processor, this technique can accurately capture the application behavior on the sliced processors. In particular, the effect of memory contention on the sliced processors can be accurately recorded in this step. Clearly, if the number of sliced processors is small, this approach will lead to savings in the amount of time required on the prototype target machine or the cycle-accurate simulator.

Cons: An important limitation of this technique is that the machine architecture of the emulating machine and the target machine should be compatible, in terms of basic data type sizes, endianness, and compiler generated paddings. This is due to the fact that the content of messages data recorded in the logs file are recorded from the emulating machine, and replayed on the target machine.

This approach naturally leads to the question of which slice(s) are to be chosen so that there is enough diversity in the collected data. This is important while modeling irregular and dynamic applications. Moreover, enough slices must be used so as to span all important SEB types and ensure that the model is accurate.

2) **Miniaturization:** An alternative approach involves running a miniature, or scaled down, version of the application at Step C_1 for the purpose of building a statistical model of execution time of each SEB. If the parameters of SEBs in the miniature application span the parameter space or allow building models that generalize well, it is possible to predict the execution times of SEBs in the target application. In other words, the miniature application must have SEBs similar to the target (full scale) application, but in smaller numbers.

Pros: This approach makes the modeling of the whole application possible on a small number of target processors or the cycle accurate simulator. In many cases, the miniature application can be chosen such that the value of parameters of its SEBs is similar to their corresponding SEB types in the target application. Thus, a robust model of SEB execution times could be constructed with limited resources.

Cons: Not all applications can be scaled down effectively. The behavior of the application may be a function of its size. In addition, the memory behavior might be very different.

This naturally leads to the questions of how to scale down an application effectively. In many scientific applications (eg. 2-D stencil, LU, BT), this can be accomplished by scaling down the input data grid. However, it may be difficult for complex and irregular applications. In Section IV-B, we demonstrate the efficacy of this approach for a real world complex application, NAMD.

3) **Random Sampling:** Instead of executing a large number of SEBs in a costly manner at Step B, it is also possible to randomly choose some SEBs to be executed, and from those results, extrapolate the execution times of all SEBs.

Pros: Random sampling is easy to implement on a cycle accurate simulator which can be dynamically switched in and out of cycle accurate mode.

Cons: Ensuring that the sampled SEBs adequately span the range of SEB parameters for all types of SEBs in the full-scale application is difficult. Implementation without a switchable cycle accurate simulator introduces significant correctness challenges.

In this paper, we implement and evaluate both slicing and miniaturization techniques.

C. Building SEB Duration Models via Machine Learning

Once the execution data for a set of annotated SEBs is collected, at Step D, standard machine learning tools are used to build models of execution times. In this paper, we use the Weka software [3] (a popular machine learning workbench) to experiment with a large class of machine learning algorithms to select the best algorithm for a given SEB type. To estimate the performance of our models, we use the standard *10 fold cross-validation* technique, where a sample of data is divided into 10 subsets. One of the subsets (called the testing set) is used to build the model while the other subsets are used to validate the model. The cross-validation process is then repeated 10 times (the folds), with each of the 10 subsets used exactly once as the validation data. The smaller the cross-validation error, the better the model. In addition, we would like our models to be simple, so as to make fast predictions of the execution times for the target machine.

We now briefly describe some of the learning algorithms/techniques that are used in the paper.

1) **Linear Regression:** uses the Akaike [1] criterion for model selection, and is able to deal with weighted instances. It has the ability to select attributes for use in the linear regression. To reduce the number of attributes that need to be collected for any SEB, we eliminate collinear attributes from the model. We select the attributes using M5's method where we step through the attributes removing the one with the smallest standardized coefficient until no improvement is observed in the estimate of the error given by the Akaike information criterion. It can also handle ill-posed problems using the Ridge regularization [16].

2) **Least median squared linear regression:** uses Linear Regression to generate linear models from random subsamples of the data. The model with the lowest median squared error is chosen as the final model [12].

3) **SVMreg:** implements the support vector machine for regression [15]. The parameters can be learned using various algorithms (e.g., RegSMOImproved [13]). There are a variety of options for the kernels to be used along with the algorithm, ranging from simple linear kernels to RBF kernels. This technique is known to generalize well and is robust to noise.

Using a synthetic benchmark called kNeighbor, we now compare the accuracy of these algorithms. kNeighbor is a program written in CHARM++. It creates a certain number of parallel objects distributed on the parallel machine; the objects are arranged in a 1-dimensional array. In each iteration, each chare element sends a message to its k neighboring objects on both sides in a wraparound fashion. When an object has received all the expected messages ($2 * K$), it does a certain amount of computation and proceeds to the next iteration. To make it more interesting, the work load of each object changes for each iteration, which is a function of parameter N: $f(N)$, N is randomly generated within a range between 1 to 100 at each iteration. The main SEB in kNeighbor benchmark does computation in proportion to N.

We collect the timing information by executing the application on 7 processors and use the data collected from one of them to build the model. The results are summarized in Table I for each of the above techniques. The second and third column show the model learnt by the algorithm and its cross-validation accuracy respectively. Subsequently, we predict the execution time of each SEB and report the relative mean error in the fourth column of the table. The relative mean error, also called relative mean difference, is the mean of the absolute difference (prediction vs. actual) divided by the mean execution time.

Method	Time(N) in μs	Cross-Val	Rel. Error
LR.	$-0.036N^2 + 0.009N^3 + 12.47$	1.83%	3.89%
LMSq.	$1.36N - 0.07N^2 + 0.009N^3 - 3.56$	1.18%	3.03%
SVM	$-2.58N + 0.0509N^2 + 0.009N^3 + 21.6$	1.07%	0.28%

TABLE I
kNEIGHBOR RESULTS ON THE THREE MODELS.

With the SEB prediction model (i.e. $time(N)$) obtained from above methods, we now simulate the kNeighbor with an increased problem size, where the range of N is now between 1 to 200, and validate the prediction with actual runs. The results are: the relative mean error in the prediction of the time consumed in SEB is 3.17% for the Least Median Square model; and 3.84% for the Linear Regression Model respectively. The SVM regression model is highly accurate even for a linear kernel, the relative mean error being 0.21%. Note that this is an example of using a scaled down run to predict larger problem size, i.e. miniaturization. In this case, the model generalizes well even outside the set of parameters of the miniature application.

Since the model is based on machine learning on a relatively small data set, the accuracy of the SEB prediction model highly depends on the quality of the data set. For example, the models investigated so far may not be able to correctly reflect the variations in delay caused by external interference, including operating system noise. However, it is possible to integrate models of noise [10] into our framework.

D. Using Modeled SEB Durations

After the models for the SEB durations are obtained, the SEB durations in the emulation log files are adjusted accordingly, as illustrated at the Step E of Fig. 2. A tool has been created for doing this. It performs the following operations for all SEBs inside a DEB: replace the duration of each SEB with a value if provided by a model; otherwise, scale the duration of the SEB by the user specified factor. The tool can be used between the emulation and simulation phases of BigSim, allowing multiple models to be evaluated with only a single, potentially costly, emulation run. After this tool has been run, the newly produced adjusted emulation logs can be fed into a full scale network simulation to predict overall performance (Step F).

IV. VALIDATION

This section offers validation case studies with NAS Parallel BT Benchmark (MPI) and a large real-world molecular dynamics application called NAMD.

A. NAS BT Benchmark

We use Abe cluster at NCSA to predict SEB execution times of BT benchmark class D, which is written in Fortran and MPI, on a target machine of 1024 cores of Ranger cluster at Texas Advanced Computing Center. Abe cluster is a cluster of 1200 Dell PowerEdge 1955 server computers, each configured with two quad-core Intel Xeon processors, 8 gigabytes of memory, and infiniBand interconnect. The Ranger cluster is an AMD Opteron cluster that is comprised of $3,936$ 16-way SMP compute nodes providing a total of $62,976$ compute cores.

Using profiling tool, we determined that the four most computational intensive subroutines in BT are: `x_solve_cell`, `y_solve_cell`, `z_solve_cell` and `compute_rhs`. The parameters we identified to represent these functions are simply the size of each major computation loop. We run BT on the BigSim emulator which emulates a 1024 core execution environment of Ranger using only 64 cores of Abe cluster. During the emulation, 8 sliced processors (out of the 1024 target processors) are chosen to record full content of each received MPI message

in order to replay. To train the prediction models, we replay the 8 sliced processors on the Ranger cluster, the target machine, one at a time, to collect the SEB execution times for building the model. Each replay is a sequential execution of the parallel BT program using the message log as input. Because Ranger and Abe have compatible processors, the trace logs can be replayed on Ranger straightforwardly.

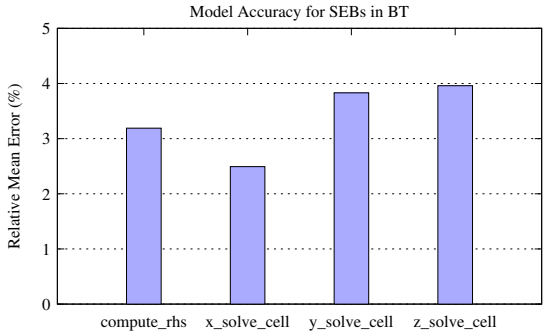


Fig. 3. SEB predictions accuracy for BT.

A linear regression model is built for each of the above SEB types and is used to predict the duration of each SEB in the class D benchmark for the 1024 Ranger processors. The prediction error is reported in Fig. 3. It can be seen that the maximum error is less than 4%, which shows that in this case, the data collected in 8 slices has enough diversity and spans all important SEB types. We now present a detailed analysis of a real-world molecular dynamics application.

B. NAMD on BlueGene/P

NAMD [11] is a scalable parallel application for Molecular Dynamics simulations written using the CHARM++ programming model. It is widely used for the simulation of biomolecules to understand their structure. In this section, we predict NAMD performance on BlueGene/P against a 4096-core run on Intrepid, an Argonne's BlueGene/P installation.

To evaluate the SEB prediction approaches, we ran NAMD with the Satellite Tobacco Mosaic Virus (STMV), a 1,066,628-atom system benchmark, with a 2AwayX configuration. This decomposition leads to approximately $149,687$ force computation objects . The force calculation performed by these objects can be categorized into seven major types: pairwise non-bonded (calc_p), pairwise non-bonded with energy (calc_pe), self non-bonded (calc_s), self non-bonded with energy (calc_se), angle, dihedral and force integration. These corresponds to the SEBs, one for each type of the force calculation. For this particular STMV benchmark, these SEBs account for more than 90% of the execution time.

First, NAMD source is modified to record SEBs and their parameters. For example, SEB "calc_p" represents the function that calculate forces on the atoms in two neighboring patches. The parameters that are chosen for it are the number of atoms in each patch and 9 other parameters which are used during the function execution. These parameters include the distance of the two patches in each dimension of the three dimensional space and the number of pairs of atoms on which different types of forces are calculated. We also record the number of inner loop iterations in the SEB. This captures the data dependent execution behavior of "calc_p".

We evaluate both slicing and miniaturization approaches of data collection and compared their accuracy.

1) Slicing: In all our experiments, the measurement data collected from the prototype is divided into two parts: training and test set. The training set (2/3 of the instances chosen randomly) is used as an input to the learning algorithm and testing set (remaining 1/3 of the instances) is used to evaluate the accuracy of the model.

When data collected from 32 processor slices is used as an input to the Linear Regression algorithm, the following model is obtained for "calc_p":

$$\begin{aligned} time = {} & 0.246 * Min_{NumAtoms} + 0.052 * Max_{NumAtoms} \\ & - 7.994 * Dx - 7.888 * Dy - 7.17 * Dz \\ & + 0.094 * Num_n + 0.013 * Num_i + 1.274 * Num_m + 19.822. \end{aligned} \quad (1)$$

Here $Min_{NumAtoms}$ and $Max_{NumAtoms}$ are the minimum and maximum among the number of atoms present in the two patches on which forces are being computed. The distance in each dimension is given by Dx, Dy and Dz respectively. Finally, the number of inner loop iterations are given by Num_n, Num_i and Num_m. The algorithm took 0.02 s to build the model and had a relative mean error of 1.77% on the testing set.

The model is intuitive as we expect the execution time to increase with the number of atoms in the two patches and the number of inner loop iterations. It is interesting to note that as the distance between the two patches increases, the number of atoms within cut-off radius decreases and hence the execution time is reduced. Clearly, Eq. 1 captures this behavior and assigns a negative coefficient to each distance term.

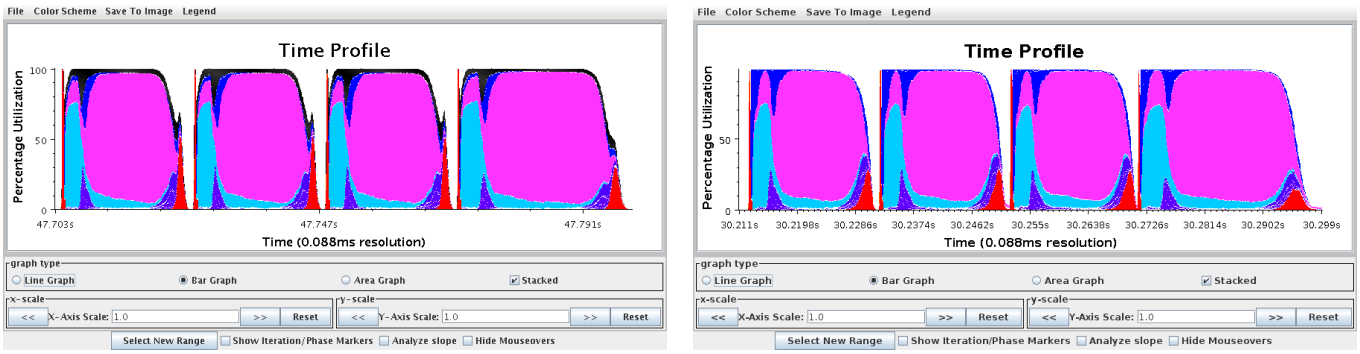
When the same data (obtained from 32 processor slices) is used as an input to the Least Median Square algorithm (with sample size 500), we obtain the following model:

$$\begin{aligned} time = {} & 0.272 * Min_{NumAtoms} + 0.068 * Max_{NumAtoms} \\ & - 9.097 * Dx - 9.131 * Dy - 8.1109 * Dz + \\ & 0.093 * Num_n + 0.013 * Num_i + 1.60 * Num_m + 13.141. \end{aligned} \quad (2)$$

The algorithm took 10.5 seconds to build the model and had a relative mean error of 1.75% on the testing set. The learning time (and the accuracy) increases as we increase the size of the random sample. Note that although the two models have different coefficients, their relative error is comparable. In comparison, the SVM regression algorithm took 49.99 s to build the model and its relative mean error of 1.7576%.

When using the slicing technique, one must be careful to ensure that the given set of slices, have instances of each SEB to be able to build a robust model. With an application like NAMD, where the mix of objects assigned to each processor varies, this is a substantial concern. Although it is difficult to theoretically ascertain the minimum number of instances required for learning, we observed that having at least 50 instances results in a robust model.

We now use the model to predict timings for all the SEBs



(a) actual run (b) prediction

Fig. 6. NAMD CPU time profile - actual v.s. prediction for STMV 4 timesteps on 4,096 cores of BlueGene/P.

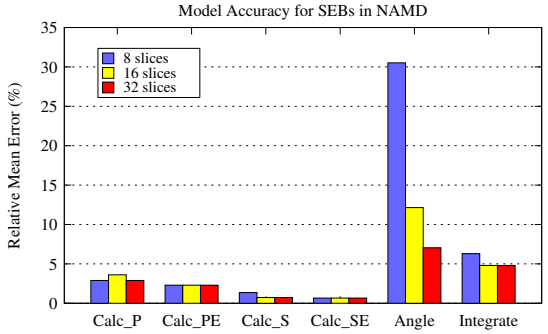


Fig. 4. SEB prediction accuracy for different number of sliced processors.

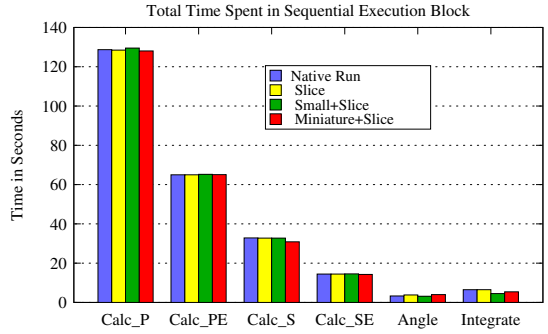


Fig. 5. NAMD SEB predictions vs actual time.

in the complete application (4096 processors). The relative mean error for several important SEBs is shown in Fig 4. The figure shows that as we increase the number of slices to collect the measurement data, the accuracy of the model improves. Note that the "calc" type SEBs consume most time of the application, and they are predicted with high accuracy (relative error less than 4%).

2) Miniaturization: In this experiment, we use a much smaller system Apoa1 (with only about 90,000 atoms) to collect the measurement data for the SEBs and use it to predict the SEB execution time of the much larger STMV system.

The linear regression model for "calc_p" is:

$$\begin{aligned} time = {} & 0.157 * Min_{NumAtoms} + 0.045 * Max_{NumAtoms} \\ & + 0.453 * Dx - 0.68 * Dy - 1.486 * Dz \\ & + 0.103 * Num_n + 0.009 * Num_i + 0.648 * Num_m + 14.555. \end{aligned} \quad (3)$$

Although the model looks quite different as compared to Eq. 1, the relative mean error in the prediction for the corresponding SEB is only 4.7%. This shows that the miniaturization technique can be used effectively for the same prediction.

We compare the accuracy of the prediction of the total time spent by the application in executing each SEB type in Fig. 5 with the native run. In the experiment "slice" we use 32 slices to build the model. In the "small+slice" experiment, we use 8 slices obtained from a smaller run (on 512 processors) of the same system, i.e. STMV. Finally, in the "miniature+slice" experiment we use 8 slices obtained from a 32 processor run of apoa1. Each of these techniques leads to a highly accurate prediction of the total time spent in the case of different "calc" type SEBs, which consume most of the time of the application. However, in the case of "angle" and "integrate", the models based on the "slice" experiment are most accurate.

Finally, to obtain trace logs for the full scale (4096 core) execution with STMV, we run NAMD emulation on only 512 cores of the machine. The trace logs from the emulation are then adjusted with the predicted SEB time using the learnt statistical model. The post-mortem parallel simulator replays these logs using a simple latency-based model to predict the performance of NAMD on 4,096 cores of BlueGene/P. The simple latency-based network model we used for BlueGene/P assumes half of its peak network bandwidth 0.22GB/s and a latency of $10\mu s$. The NAMD startup phases are fast-forwarded, since it is not our main interest. The simulation runs for total of 8 timesteps, and we predict the last 4 steps in full details.

Using the model obtained from the slicing method above, a per step time of $21.7ms$ is predicted, v.s. the actual time per step of $21.1ms$. To further analyze the details of the prediction, we ran NAMD with Projections performance tool and compare the application behavior to the predicted results using the time profile tool. The Projections time profile view, as illustrated in Fig. 6, shows CPU utilization in percentage summed over 4,096 cores over each interval, each SEBs is shown in different colors. Due to the overhead with performance data instrumented for Projections at runtime, the actual run is slowed down by about 10%. Nevertheless, we can still see that the predicted NAMD behavior over time matches well with the actual run on the 4,096 cores of BlueGene/P, with only a little inaccuracy at the end of each time step, probably due to the latency model not capturing contention effects.

V. Related Work

Machine learning techniques have been used to construct models of overall application execution times based on sparsely sampled data on multiple platforms for large multi-dimensional parameter spaces [14]. The BigSim techniques presented in this paper differ because they use machine learning techniques to build individual SEB performance models, instead of whole program models. Additionally our approach can capture detailed behavioral characteristics of the application or simulated machine rather than just the total application execution time. Various sampling techniques are also often used in the simulation of micro-architectures [17].

Recently researchers involved with the COTSon project, which attempts to simulate clusters of hundreds of nodes, have also found that full always-on cycle-accurate whole system simulations are cost prohibitive, and thus sampling and slicing approaches are useful [2]. The traditional slicing performed in COTSon and other systems is the choosing of regions of a serial program to execute at a detailed micro-architecture level, not slicing the set of processors [7]. Because the focus of the BigSim project is to simulate scientific applications on hundreds of thousands of nodes, this paper proposes the use of a novel type of slicing and miniaturization technique.

In the field of task scheduling, it has been shown that machine learning techniques such as neural networks can be used to construct performance models of the tasks that comprise a parallel application. The models can then be used within a task scheduler to produce better overall schedules [8].

VI. Conclusion

Predicting sequential execution blocks of a large scale parallel application is an essential part of accurate prediction of the overall performance of the application. However, it is a difficult task due to the complexity of the application itself, compounded by excessively simulation slowdown and insufficient resource availability.

In this paper, we propose an approach based on statistical models to accurately predict the performance of significant sequential execution blocks. The approach consists of several steps from emulation to post-moterm simulation to make it possible to predict performance of very large scale applications. This is done by first scaling down the simulation problem size to be manageable using slicing and miniaturization techniques to capture the characteristics of the application. Using machine learning techniques, the knowledge gained is then turned into a statistical model for each SEB, which can be used to predict the application at full scale running on the emulator. We deployed these techniques in a trace-driven simulation framework to capture the whole behavior of the application and predict the overall performance. The validation results with both synthetic benchmarks and NAMD show that the proposed approach is effective and accurate. For the upcoming Blue Waters machine, we plan to use these techniques, together with a detailed contention-based network simulation for Blue Waters interconnect to simulate several NSF-chosen applications including NAMD and predict their performance on the machine.

Acknowledgments

The research component of BigSim has been supported by NSF awards NGS-0103645 and CSR-SMA-0720827, whereas the BigSim deployment for Blue Waters is being funded by NSF via the Blue Waters project, under grant OCI-0725070.

References

[1] H. Akaike. A new look at the statistical model identification. *IEEE Transactions on Automatic Control*, pages 716–723, 1974.

[2] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. COTSon: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[3] S. R. Garner. Weka: The waikato environment for knowledge analysis. In *In Proc. of the New Zealand Computer Science Research Students Conference*, pages 57–64, 1995.

[4] F. Gioachin, G. Zheng, and L. V. Kalé. Robust Record-Replay with Processor Extraction. In *Proceedings of the Workshop on Parallel and Distributed Systems: Testing, Analysis, and Debugging (PADTAD - VIII)*, Trento, Italy, July 2010.

[5] J.L.Peterson, P.J.Bohrer, L.Chen, E.N.Elnozahy, A.Gheith, R.H.Jewell, M.D.Kistler, T.R.Maeurer, S.A.Malone, D.B.Murrell, N.Needel, K.Rajamani, M.A.Rinaldi, R.O.Simpson, K.Sudeep, and L.Zhang. Application of full-system simulation in exploratory system design and development. *IBM Journal of Research and Development*, 50, 2006.

[6] L. V. Kale, G. Zheng, C. W. Lee, and S. Kumar. Scaling applications to massively parallel machines using projections performance analysis tool. In *Future Generation Computer Systems Special Issue on: Large-Scale System Performance Modeling and Analysis*, volume 22, 2006.

[7] T. Lafage and A. Seznec. Choosing representative slices of program execution for microarchitecture simulations: a preliminary application to the data stream. pages 145–163, 2001.

[8] J. Li, X. Ma, K. Singh, M. Schulz, B. R. de Supinski, and S. A. McKee. Machine learning based online performance prediction for runtime parallelization and task scheduling. In *IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 89–100, April 2009.

[9] C. Mei. A preliminary investigation of emulating applications that use petabytes of memory on petascale machines. Master's thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 2007. http://charm.cs.uiuc.edu/papers/ChaoMeiMSThesis07.shtml.

[10] F. Petrini, D. Kerbyson, and S. Pakin. The Case of the Missing Supercomputer Performance: Achieving Optimal Performance on the 8,192 Processors of ASCI Q. In *ACM/IEEE SC2003*, 2003.

[11] J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: Biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE conference on Supercomputing*, 2002.

[12] P. J. Rousseeuw and A. M. Leroy. *Robust Regression and Outlier Detection*. John Wiley & Sons, second edition, 2003.

[13] S. Shevade, S. Keerthi, C. Bhattacharyya, and K. Murthy. Improvements to the smo algorithm for svm regression. *IEEE Transactions on Neural Networks*, pages 1188–1193, 2000.

[14] K. Singh, E. İpek, S. A. McKee, B. R. de Supinski, M. Schulz, and R. Caruana. Predicting parallel application performance via machine learning approaches. *Concurr. Comput. : Pract. Exper.*, 19(17), 2007.

[15] A. J. Smola and B. Schoelkopf. A tutorial on support vector regression. *Statistics and Computing*, pages 199–222, 2004.

[16] Tychonoff. *Solution of Ill-posed Problems*. Winston and Sonses, Washington.

[17] J. J. Yi, S. V. Kodakara, R. Sendag, D. J. Lilja, and D. M. Hawkins. Characterizing and comparing prevailing simulation techniques. In *HPCA '05: Proceedings of the 11th International Symposium on High-Performance Computer Architecture*, 2005.

[18] G. Zheng, A. K. Singla, J. M. Unger, and L. V. Kalé. A parallel-object programming model for petaflops machines and blue gene/cyclops. In *NSF Next Generation Systems Program Workshop, 16th International Parallel and Distributed Processing Symposium(IPDPS)*, April 2002.

[19] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé. Simulation-based performance prediction for large parallel machines. In *International Journal of Parallel Programming*, volume 33, 2005.