Application Modeling for Scalable Simulation of Massively Parallel Systems

Eric Anger and Sudhakar Yalamanchili
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, Georgia
{eanger,sudha}@gatech.edu

Damian Dechev, Gilbert Hendry, and Jeremiah Wilke
Sandia National Laboratories
Livermore, California
{ddechev,ghendry,jjwilke}@sandia.gov

***Abstract*—Macro-scale simulation has been advanced as one tool for application–architecture co-design to express operation of exascale systems. These simulations approximate the behavior of system components, trading off accuracy for increased evaluation speed. Application skeletons serve as the vehicle for these simulations, but they require accurately capturing the execution behavior of computation. The complexity of application codes, the heterogeneity of the platforms, and the increasing importance of simulating multiple performance metrics (e.g., execution time, energy) require new modeling techniques.**

We propose flexible statistical models to increase the fidelity of application simulation at scale. We present performance model validation for several exascale mini-applications that leverage a variety of parallel programming frameworks targeting heterogeneous architectures for both time and energy performance metrics. When paired with these statistical models, application skeletons were simulated on average 12.5 times faster than the original application incurring only 6.08% error, which is 12.5% faster and 33.7% more accurate than baseline models.

I. Introduction

Both the scientific and enterprise computing domains are seeing the rise of extreme scale systems to handle explosive growth in problem sizes. These systems will need to be designed, without the luxury of prototyping, to fit within the tightening constraints of power, heat dissipation, memory bandwidth, and processor speed. We foresee hardware–software co-design taking a key role in their development, specifically the use of scalable co-simulation of application codes and system models.

In this paper we are concerned with a particular methodology for hardware–software co-design referred to as *macro-scale simulation*, where candidate applications and hardware configurations are simulated at the scale of hundreds of thousands of cores to understand the impact of system design at scale. Inter-node communication is modeled as flows on the network, which is analyzed for congestion behavior and its impact on system performance. Such simulations have demonstrated high scalability [1], capable of reaching tens of thousands of nodes. However to ensure high-fidelity simulations, these abstracted communication models must be coupled with fast, accurate end-point computation models.

Developing accurate models of applications on a compute node is a complex task [2]. Cycle-level hardware simulation is infeasible at scale. We need more abstract models of the *effects* of computation on the system, in particular the time and energy consumed, rather than the *mechanism*. In this paper we propose a modeling methodology based on the notion of *application skeletons*; these are application implementations where regions of code (e.g., compute-intensive loop nests) can be replaced by analytic models of their physical properties (e.g., execution time and energy). These models can be used to determine the timing and duration of network traffic injection or to explore the time-varying distribution of power consumption.

To ensure that the macro-scale system simulators retain their scalability when running application skeletons, the models of computation must evaluate quickly. The baseline model, based on loop iteration counts, requires manual tuning and is limited by its simple structure, resulting in high error. We propose the use of statistical modeling, which learns the relationships between application-level input parameters and performance through analysis of instrumentation data taken from the application running on real hardware. This modeling technique can flexibly handle model construction for complex architectures and applications while reducing domain expertise required by users building the application skeletons. This method is used across execution models and language selection to predict both execution time and energy. This work uses the Eiger Statistical Modeling Framework [3] for model generation, augmented with greater flexibility in selecting input metrics and regression functions.

The main contributions of this work are:

- the addition of computation models to application skeletons, permitting high-fidelity simulation at scale,
- a technique for converting an application into a skeleton, and
- the use of statistical models, as generated by Eiger, to predict how different input parameters translate into execution time and energy.

This paper is structured as follows: Section II describes previous work in application modeling and simulation. Section III presents application skeletons. Section IV illustrates the model generation process. Section V describes a technique for construction application skeletons. Section VI contains

a demonstration of the models' performance, as well as implications for simulation at scale, followed by concluding remarks in Section VII.

II. Prior Work

Several simulators have been developed to generate performance estimates for high performance computing architectures. Simulators range from high-fidelity and computationally expensive simulators for measuring performance between two nodes [4], [5] to lower-fidelity and lower-cost simulators that can estimate performance on large-scale machines[1]. Often emphasis is on MPI emulation or simulation, with prominent simulators including BigSim [6], SIMGRID [7], and PSINS [8] in addition to SST/macro [9] used here. Similar to our work, there are notable examples of simulators that fit application runtime to analytical models [10]. Our work models individual compute regions rather than the whole application.

Simulation Techniques Three well-known approaches have been investigated for estimating large-scale performance. The most common approach is direct execution of the full application on the target system [11], [12], [6]. This simulation approach uses virtual time unlike normal benchmarking that uses real time. Performance is measured by using a processor model and communication work in addition to simulated time for a modeled network.

Another approach requires tracing the program in order to collect information about how it communicates and executes [6]. The resulting trace file contains computation time and actual network traffic. Time-independent trace replay based on hardware performance counters extrapolated to multiple architectures has been demonstrated [13]. Still, tracing does not scale to a different number of processors or new problem sizes.

A third approach is to implement a model skeleton program as a simple, curtailed version of the full application but complete enough to simulate realistic activity [14], [15]. This approach has the advantage that the bulk of the complex computation can be replaced by simple timing information. The skeleton application provides a powerful method for evaluating the scalability and efficiency over various architectures of moderate or extreme scales.

Skeleton-driven Simulation The use of kernels or miniature applications in performance analysis is well established. Two well-known collections of kernels are the NAS Parallel Benchmarks (NPB) [16] and the PARKBENCH suite [17]. The kernels in these suites represent common computational patterns that are found in many full-scale applications. Examples include sorting algorithms, the Fast Fourier Transform, and matrix-based numerical algorithms. While these suites provide simpler implementations of important algorithms than full scientific codes, they represent generic algorithms that lack any nuances that would be found in specific application implementations. Some approaches investigate automatically synthesizing skeletons from communication traces [18], [19], but this requires extensive trace collection and may not capture behavior produced with extrapolated application parameters outside the calibration range. Static analysis techniques have been used to identify computations that have no impact on control flow or communication and replace them with symbolic estimates for time [14]. Automated methods have been demonstrated for transforming scientific codes through the use of both static analysis and runtime information in the form of MPI communication patterns [20], [21].

Statistical Modeling Various techniques from the field of machine learning have been applied to high performance computing. Applications have been characterized using principal component analysis [22] and hierarchical clustering [23] in order to describe the differences between applications in benchmark suites as well as to indicate how applications stress different parts of the hardware pipeline. Other studies [24] look into statistical models for design space exploration so that only a small sampling of machine configurations must be simulated. One technique used to predict execution time, power, and energy consumption is Artificial Neural Networks (ANNs) [25], [26]. While this method provides high quality predictions, it obscures the meaning and interpretation of the model while prohibiting the designers from informing the modeling process. Some works [27] use linear regression models similar to Eiger to predict performance and energy, but look at the entire execution of the application. In contrast, this work only predicts regions of computation using statistical models as a component of the application skeletons, which allows for flexibility in simulating applications across machine topologies and configurations.

III. Application Skeletons

Simulating the execution of applications on a large number of cores requires us to abstract application software with minimal compromises in simulation accuracy. Recently we have seen the development of *macro-scale simulation* models, in contrast to cycle-level or micro-scale simulation models. The macro-scale simulation models are driven by *application skeletons*: a full application, modified to retain its control flow and structure, but with code segments replaced by analytic models to compute the physical resources—such as execution time or energy—consumed by the corresponding code segment. In general the *effect* of execution characteristics are modeled rather than the *mechanism*, i.e., the procedure a code goes through during execution. The use of application skeletons enables modeling to be applied at a larger scale that otherwise feasible.

This work leverages the SST/macro simulator, one branch of the SST project [9], for coarse-grained macro-scale simulations. SST/macro provides the foundation for large scale systems research and has previously been used in high performance network-related studies [28], [29]. The general organization of SST/macro is shown in Figure 1. The application skeleton is comprised of communication calls (e.g., MPI, HPX, SHMEM) and models of computation. Control flow structure is preserved. This paper seeks to improve the models of computation, including the addition of energy models.

Application skeletons serve as a more accurate (from an execution perspective) representation of the time and energy consequences of application execution.

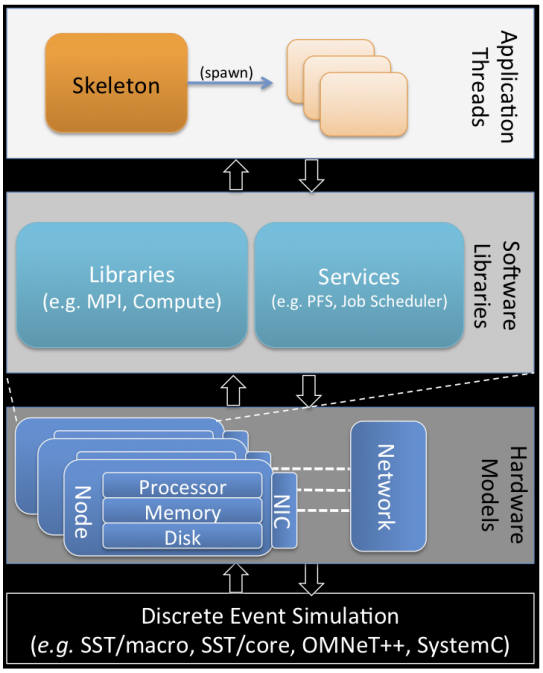

Fig. 1: The structure of the SST/macro simulator.

```
for(int iteration = 0; iteration < MAX_ITERS; ++iteration){
   local_computation();
   if(iteration % SYNC_PERIOD == 0){
      local_aggregation_step();
      global_sync();
   }
}
```

(a) Original application.

```
for(int iterations = 0; iteration < MAX_ITERS; ++iteration){
   // REMOVED: local_computation();
   if(iteration % SYNC_PERIOD == 0){
      // REMOVED: local_aggregation_step();
      model_communication(global_sync);
   }
}
```

(b) Skeleton application with communication calls replaced with a model, while retaining correct control flow.

```
for(int iterations = 0; iteration < MAX_ITERS; ++iteration){
   model_computation(local_computation);
   if(iteration % SYNC_PERIOD == 0){
      model_computation(local_aggregation_step);
      model_communication(global_sync_model);
   }
}
```

(c) Skeleton application with the inclusion of both communication and computation models.

Fig. 2: Creation of a compete application skeleton from its original.

The timing/energy behavior of application skeletons should be as close to the original application as possible; any deviations degrade the accuracy of the simulation. The skeletons should retain control flow characteristics as close to that of the original application as possible. Take for example the simplified structure of an extreme-scale application as shown in Figure 2a, where local and global work is performed on a periodic basis. In this application, computation is performed locally several times before synchronization across the system. For a skeleton to accurately reflect the behavior of its source application, it must retain the control flow and model the execution effects of computation regions.

The SST/macro simulator replaces all communication directives with its own models. Such a skeleton, shown in Figure 2b, would correctly capture the number and duration of all communication functions. However, it misses the execution behavior of potentially lengthy computation between communication calls; the behavior of this application may depend on the loads of each processor and how well the synchronization step is performed. The work in this paper enhances SST/macro with *compute models*, taking the place of all code segments within the skeleton application, as shown in Figure 2c. These models estimate the consumed time or energy taken by a computation region.

A. Data-Dependent Computation

Ensuring correct flow of the application skeleton is important when considering computation that is data-dependent; there may be computation regions whose execution is prescribed by the data being processed. These types of computations manifest in areas such as iterative algorithms, where termination is based on convergence criteria, and in sparse data representations, where meta-data such as dimensionality is not fixed.

There are several approaches that can be used to ensure that the execution behavior of a skeleton matches its parent application. The most direct, but most costly route, is to preserve in the skeleton any calculations that dictate the termination conditions of execution. A typical manifestation of this is when some internal data structures must be initialized before computation is performed on them. The exact sizes of these structures are not known until after the initialization has completed; here their construction is permitted, a time- and resource-intensive process, to access these dimensions.

When data-dependent computation is performed throughout the application execution, more care must be taken. An example is when the loop iteration count is not known statically, such as operating on sparse matrix representations. When these matrices are multiplied, only the nonzero elements are iterated through. A trade-off must be made here between accuracy of the simulation and runtime expense in maintaining execution-dependent information. Such an analysis is beyond the scope of this work.

IV. Compute Modeling

Making models of computation begins with finding the places in the application skeleton where computation takes

place. Typically this involves nested loops or other forms of iterative computational load, such as found in matrix algebra. Standard performance profilers like `gprof` are invaluable for this task; they break down where time is spent in the application and the function call hierarchy. We have found that converting the most time-consuming functions in the profile into models works well. These regions should avoid containing network communication calls.

We propose a method for generating high-quality analytic models for these regions through statistical inference. A corpus of execution data from instrumented executions of the application, including source-level parameter values, is used as training data to construct an analytic model of execution time or energy. This modeling approach provides many benefits over traditional approaches to constructing analytic models, including removal of the need for deep domain expertise (e.g., in energy consumption) during model construction, increase in the diversity of what can be modeled, and significant reduction in manual tuning. This work builds upon the Eiger Statistical Modeling Framework to generate statistical models for execution time and energy.

Eiger is based on a methodology for constructing statistical models from instrumentation data. Eiger provides a standardized infrastructure and API for adding instrumentation to applications, a relational data store for collecting and managing that data, and a workflow for the creation and analysis of models. The main goal of the Eiger project is to facilitate intelligent, semi-automated model construction without requiring expert knowledge of hardware or software. Instead of meticulous domain analysis, the models are learned from the measurement data. The infrastructure is designed for flexibility in adding new parameters controlling model construction and trading accuracy for performance.

The framework does not predefine what should constitute an input parameter or a performance metric. Input parameters may include hardware (e.g., cache size, cache line size, register file size) and software features (e.g., input data size, stencil size, DMA block size) while performance metrics are any measurable metric (e.g., execution time, energy, failure rates). The general procedure for model construction can be seen in Figure 3 and consists of three main components: Principal Component Analysis, Clustering, and Forward Model Selection.

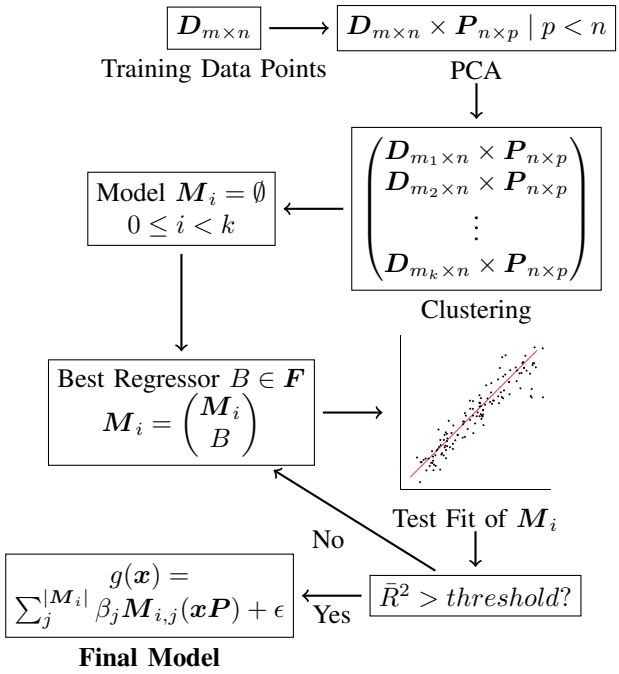

Fig. 3: The steps of model construction.

A. Principal Component Analysis

Before any modeling takes place, Eiger constructs a single, large matrix $\boldsymbol{D}_{m\times n}$ representing all instrumentation data. Each column is a different input parameter and each row is a separate data point corresponding to a unique application execution. In its effort to minimize domain expertise to construct models, Eiger recommends liberally including input parameters, even if they ultimately will not be used in the model. This step, Principal Component Analysis (PCA) [30], aims to compress the number of columns, eliminating unnecessary data and simplifying model generation. A linear transformation $\boldsymbol{P}_{n\times p}$ is produced, converting the set of possibly correlated input parameters into a derived set of uncorrelated parameters. Derived parameters with low variance can be eliminated. A concern [31] with this technique is that even parameters exhibiting low variance may relate strongly to the dependent variable, and would be valuable information to discard. For this reason, PCA is leveraged to eliminate parameters with *zero* variance, simplifying the data set without throwing away information.

B. Clustering

The next modeling step is Clustering, where similar data points (i.e., rows of the output matrix from the PCA step) are grouped together. Rather than using all the data points in the model fitting step, each cluster goes through fitting separately, under the assumption that this composite of fits will perform better [32]. Eiger uses *k-means* clustering, where a data point belongs to the nearest of k clusters. The center point of the cluster is defined as the mean of the input parameter values for all the data points in that cluster. Eiger performs the next step of model construction for each of the clusters, creating k different analytical expressions. When evaluating the model, the closest cluster must be found and its associated analytical expression used. Choosing the value for k is left to the user; this analysis is outside the scope of this work.

C. Forward Model Selection

The last transformation fits an analytical expression of the input parameters in a cluster to calculate the result metric. This is achieved with a linear combination of functions, called *regressors*, that are applied to the input parameters. These regressors come from a pool $\boldsymbol{F}$ of candidates. In the Eiger framework a selection approach is taken to progressively

include regressors in the final model only if they improve the quality of the fit. This approach is *forward* in the sense that it begins with an empty model and adds regressors to it, in contrast with a *backward* approach in which the model starts out with every possible function and unsatisfying elements are removed.

Beginning with an empty model, regressors are chosen from the pool and added one by one to select the one that increases the quality of the fit the most. The coefficient of determination (R^2) gives a measure of how well a model is able to map predicted values to their associated training value; it ranges from 0, indicating no correlation between the prediction and the observed valued, to 1, indicating exact replication by the model. Eiger uses the adjusted coefficient of determination ($\bar{R}^2$), a modification of R^2 taking into consideration the number of terms in the model [33]. $\bar{R}^2$ will decrease if the added regressor results in a model that performs worse than would be expected by adding a random regressor. To control over-fitting, in which the predictability of new data points is sacrificed in order to ensure the model approaches the training data as closely as possible, the winning regressor is only added if the amount by which it increases the fit is larger than a user-provided threshold. In this work, selecting the threshold value is an empirical procedure. Model construction finishes when there are no more models left in the pool or when the winning regressor does not surpass the threshold. The final model is comprised of each selected regressor $\boldsymbol{F}_i$ and an associated weight β_i, plus an error term ϵ.

We found that there are instances where the user has some intuition about the relationship between input parameters and performance and wish to generate a model based on this intuition, (i.e., based on a specific set of regressors and input parameters). For example, the user may wish to construct a linear model or a model that is logarithmic in the input data set size. Consequently, we have extended Eiger to enable the user to *a*) specify the set of regressors to be used, and *b*) specify the set of input parameters to be used. For example, even though the set of input parameters is quite extensive, the user may wish to construct a model based on two input parameters and two specific regressors.

V. Skeletonization Procedure

The original source code of the candidate application is augmented so that compute intensive code regions can be easily replaced by the model thereby producing an application skeleton. These modified applications are then executed by linking with the SST/macro simulator that simulates the execution of this application on a large scale parallel architecture model. To facilitate the collection of measurement data, we have developed the Lightweight Performance Data Collectors (`lwperf`) tool[1], a collection of simple, portable macros aimed at making it easy to gather high-level algorithm features and performance numbers from real applications. A single code markup scheme is provided which, based on compilation flags, can record performance data for code regions from parallel application execution. The performance data is recorded in an Eiger database. `lwperf` supports C, C++, and modern Fortran. Figure 4 shows the different aspects of model generation, the typical procedure for which is:

1) Place appropriate `lwperf` initialization (`PERFINIT`) and finalization (`PERFFINALIZE`) calls at the beginning and end of the application, respectively.
2) Locate the compute regions to be modeled. Typically this involves nested loops or accelerator kernels, but can be at user-defined granularity.
3) Surround the compute regions with appropriate calls to `lwperf` macros `PERFLOG` and `PERFSTOP`, indicating a region name as well as any parameters which may be important determinants of the performance of this region, such as loop bounds.
4) Compile the application with the data collection flags set and run the program over various problem sizes. This will fill the Eiger database with training data.
5) Run Eiger model generation for each compute region, as described in Section IV.
6) Rebuild the application with the simulation flags set. During execution, the marked compute regions will be replaced with with calls to the models, skeletonizing the application.
7) Simulate the skeleton with SST/macro. This requires linking with SST/macro and providing a configuration file describing the machine model to use. More details on the changes needed to make an application skeleton run in SST/macro can be found in its documentation[2].

Handling heterogeneous or parallel node architectures is straightforward with Eiger models. Eiger makes no assumptions about the execution model, instead learning the relationships between input parameters and performance for each marked code region. As long as the entire computation region is captured and profiled, including any intra-node communication, Eiger will attempt to learn a model of performance from the data.

At any code region where data collection is desired, the collection point is defined by a unique name and the set of parameters characterizing the workload. Figure 4a shows the same function from Figure 5a with `lwperf` instrumentation calls inserted. The region is uniquely identified by the first argument and indicates two metrics of interest, the matrix dimensions `M` and `N`. A single data point is then recorded every time this region of the program is entered, along with the time elapsed as shown in Figure 4b.

To facilitate skeletonization, `lwperf` can be configured to remove and replace all instrumented code with calls to Eiger models for running within the SST/macro simulator. Each collection point is replaced with calls to an Eiger model under the assumption that each characterizing parameter is

[1]The source code and documentation for `lwperf` is available at https://github.com/gtcasl/lwperf

[2]The source code and documentation for SST/macro is available at https://bitbucket.org/sst-ca/sstmacro.

```
void foo(int M, int N, int** matrix){
   PERFLOG(foo, M, N);
   for(int i = 0; i < M; i++){
      for(int j = 0; j < N; j++){
         int temp = (matrix[i][j] + i) % 10;
         matrix[i][j] += temp; } }
   PERFSTOP(foo, M, N);
}
```

(a) Annotated Original

```
void foo(int M, int N, int** matrix){
   double start = MPI_Wtime();
   for(int i = 0; i < M; i++){
      for(int j = 0; j < N; j++){
         int temp = (matrix[i][j] + i) % 10;
         matrix[i][j] += temp; } }
   double stop = MPI_Wtime();
   EIGER_record("foo", M, N, stop - start);
}
```

(b) Data Collection

```
void foo(int M, int N, int** matrix){
   /*
   for(int i = 0; i < M; i++){
      for(int j = 0; j < N; j++){
         int temp = (matrix[i][j] + i) % 10;
         matrix[i][j] += temp; } }
   */
   double foo_time = EIGER_compute_time("foo", M, N);
   SSTMAC_compute_time(foo_time);
}
```

(c) Skeleton

Fig. 4: Transformation of a simple function into a compute model. The original application is transformed into data collection and skeleton versions during compilation.

used as an input feature to the model. Figure 4c shows how computation is replaced with a call to the model for execution time. This technique allows for the application to serve as both the instrumentation source and the skeleton.

VI. Experimental Results

To demonstrate the benefits of a comprehensive modeling tool like Eiger, we construct a simple model by hand that tries to replicate the structure of loop nests. Computation is represented by a linear function of the loop bounds scaled by a *loop factor* describing the amount of "work" performed per iteration. For example, the loops in the simple function in Figure 5a might be replaced with a single call such as in Figure 5b, specifying the nested lower and upper loop bounds, and in this case, a loop factor of 2. The product of the loop bounds and loop factor is scaled by a set of empirically derived constants to provide an estimate of the performance metric.

The loop model is simple and intuitive. However, it is a manual process and can only represent linear relationships. Therefore it is not easily adapted to the parallel implementations of nested loops or execution on heterogeneous architectures where performance is a more complex function of many interacting microarchitectural features. In this paper it serves as a baseline for comparison for cases where such simple models would suffice.

```
void foo(int M, int N, int** matrix){
   for(int i = 0; i < M; i++){
      for(int j = 0; j < N; j++){
         int temp = (matrix[i][j] + i) % 10;
         matrix[i][j] += temp; } }
}
```

(a)

```
void foo(int M, int N, int** matrix){
   /*
   for(int i = 0; i < M; i++){
      for(int j = 0; j < N; j++){
         int temp = (matrix[i][j] + i) % 10;
         matrix[i][j] += temp; } }
   */
   SSTMAC_compute_loops2(0, M, 0, N, 2);
}
```

(b)

Fig. 5: Example function (a) and the same function where the computation is replaced by the loop model (b).

A. Experimental Setup

We use four scientific application codes familiar to the HPC community to demonstrate adding Eiger models to application skeletons:

1) **MiniMD** is a molecular dynamics mini-application from the Mantevo project [34], created to investigate improving spatial-decomposition particle simulations as a simpler, more accessible version of LAMMPS [35]. Parameters to miniMD include problem size, atom density, temperature, time step size, number of time steps, and particle interaction cutoff distance. This application offloads computations to GPU accelerators using the CUDA parallel programming language [36].
2) **HPCCG** is a simple conjugate gradient benchmark code for a 3D chimney domain, also from the Mantevo project. It generates a 27-point finite difference matrix with a user-prescribed sub-block size on each processor, and weak scales well to a large number of processors.
3) **MiniFE** is a proxy application from the Mantevo suite for unstructured implicit finite element codes. It is similar to HPCCG but provides a much more complete vertical covering of the steps in this class of applications. This application also uses CUDA for GPU acceleration.
4) **LULESH** discretely approximates explicit hydrodynamics equations [37] found in complex software packages like ALE3D by partitioning the spatial problem domain into a collection of volumetric elements defined by a mesh. This application uses OpenMP [38] to parallelize computation across CPU threads.

Constructing Eiger models relies upon a corpus of training data taken from real executions, including the performance metrics associated with each region of computation. We measure application energy consumption for CPU-only applications using the Running Average Power Limit (RAPL) hardware performance counters available on recent Intel processors. These counters are measured using the Performance Ap-

TABLE I: Machine configurations for collecting time and energy training data.

Use	Parameter	Value
time	CPU	Intel i7-920
time	CPU threads	8
time	CPU clock rate	2.67 GHz
time	Memory size	6GB
time	GPU model	NVIDIA GTX 660 Ti
time	GPU cores	1344
time	GPU clock rate	915 MHz
time	GPU memory size	2GB
energy	CPU	Intel i7-4770
energy	CPU clock rate	3.40 GHz
energy	CPU threads	8
energy	Memory size	16GB

plication Programming Interface (PAPI) [39] and cover energy consumption for the entire processor package. Infrastructure for measuring energy consumption of the entire node (i.e., memory modules, accelerators, network interface components) can be used to better describe the energy characterization of a node, but is outside the scope of these experiments. The systems used for these experiments are described in Table I.

B. Model Fit

Here we examine how closely the models predict the training data. For these experiments, each application was executed unmodified for several different input sizes, using the data collection procedure described in Section V. This training data was then used to generate the models; one model is generated for each region of computation.

The improved fit of the Eiger models over the baseline loop models can be clearly seen in the region of computation in MiniFE where the Dirichlet boundary conditions are imposed; this region requires calls to two GPU kernels. The fits of the models are shown in Figure 6. The loop models were hand-tuned until they converged as close as possible to the training data. This process required significantly more time to construct, requiring a thorough examination of the original application source code. Even then, the performance of these models was in general poorer than Eiger. The loop model can statically subdivides the number of elements on the boundary across all the GPU threads, achieving an R^2 value of 0.912 and an average error of 6.73%. In contrast, Eiger generates a model that scores an R^2 value of 0.983 and an average error of 2.43%, without manual intervention.

As this computation is performed on the GPU, factors such as how long it takes to transfer all the data to the device, how work is assigned to threads, and how those threads get scheduled on the device at runtime contribute to variability in performance unrelated to the size of the vectors. To increase the quality of the loop models, more rigorous attention to the inner workings of the device could be harnessed, but would require expert knowledge of the architecture.

C. Simulation Accuracy

One concern with the models is that they may be too strongly tied to the training data. This phenomenon is called

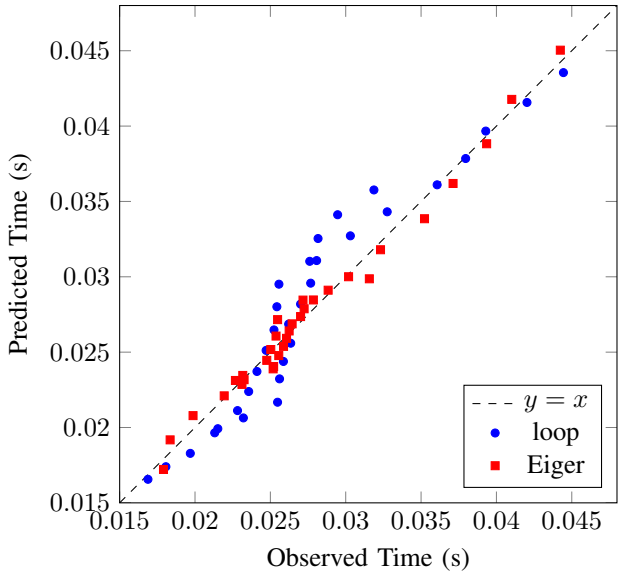

Fig. 6: Observed training values and their associated loop model predictions for MiniFE Dirichlet boundary condition region of computation.

overfitting, indicating that the model is biased towards the training data. While a model may be a strong predictor of the training data, there might be ancillary effects that go unobserved as the problem size changes. One example of this happens with the region of computation in HPCCG where two vectors are scaled and added together. For smaller problem sizes, the vector length strongly correlates with execution time. However as the vectors get larger, the structure of the memory hierarchy affects the average access latency per element. This causes the loop model to diverge from the training data.

In contrast, the model generation process in Eiger avoids overfitting through use of a threshold parameter. This tunable parameter prevents the inclusion of elements in the model that would result in overfitting. Figure 7 shows the effect of the threshold value (i.e., how biased the model is to the training data) on the average number of regressor functions in the model and the resulting simulation error. There is a middle ground where increasing the threshold no longer reduces overfitting but rather begins to eliminate necessary model terms, increasing overall error.

With the models trained, we now show how the individual models interact when simulating an entire application skeleton. Having accurate application skeletons, across input problem sizes, forms the foundation for macro-scale simulation. To eliminate any error caused by the communication models in SST/macro, we only simulate a single node. This test demonstrates that all computation performed by the application is modeled and to guarantee that any code removed during the skeletonization process does not change the overall structure of the application. Figure 8 illustrates how close the simulations are to correctly predicting the execution times of all four applications and the energy consumption of HPCCG

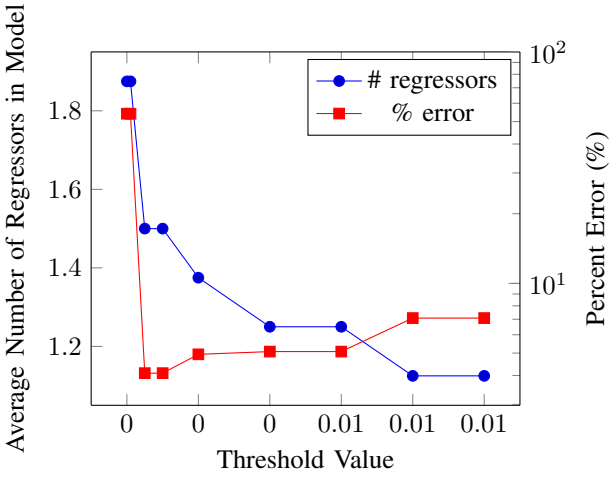

Fig. 7: Effect of threshold value on overfitting for HPCCG.

and LULESH. On average, the models generated by Eiger result in 6.08% error, where the loop models incur a 9.17% error.

D. Simulation Speedup

SST/macro has been demonstrated to simulate parallel systems with tens of thousands of nodes in minutes [1]. To enable scalable macro-scale simulation, the models of computation added to application skeletons must be fast. These experiments compare the execution time of the original application with its application skeleton. Where as testbed systems and cycle-level simulators tend to run orders of magnitude slower than on the native machine, simulating our application skeletons results in a *speedup* of simulator runtime over the native execution time of the application, as shown in Figure 9. These speedups tend to increase with the problem size; the predicted computation time increases as the problem size increases, but the time it takes to poll those models remains constant, resulting in an overall increase in the speedup of simulator runtime.

Eiger models are capable of increased simulator speedup over loop models due to their higher level of abstraction; for similar accuracy, more granular loop models are required. Take, for example, creating a loop model for the execution time of a GPU kernel. Memory transferred from the host to the device is represented by the amount of data to be transferred and the time to transfer an individual unit. The computation in the kernel is modeled, requiring a non-trivial understanding of how tasks are scheduled and the breakdown of tasks to be performed.

This is significantly easier to model with Eiger: the entire kernel can be abstracted into a single model that is learned from training data, encompassing all aspects of execution of the kernel, including data transfers. This reduces the total number of times the model needs to be polled, decreasing the running time of the simulation. This is the case for the MiniMD, MiniFE, and LULESH skeletons. The HPCCG skeleton behaves differently; simulation speedup is lower when using Eiger models compared to loop models. This is due to the simplicity of computation regions within the application; there are few opportunities for abstracting large regions of computation into Eiger models, resulting in a similar number of calls to Eiger models as loop models. Eiger models are more computationally expensive to evaluate than loop models, resulting in lower speedup. On average, the skeletons with loop models ran 12.1 times faster than the original application and skeletons with Eiger models ran 12.5 times faster.

VII. Conclusion

Co-design is enabled by high fidelity macro-scale simulation, providing software writers and hardware architects the ability to reason about the way exascale systems operate at scale. Application skeletons, the vehicles of macro-scale simulation, require high-fidelity models of computation. In this paper we presented a method for modeling time spent in compute regions: statistical models that learn the relationship between input parameters and the performance metric (e.g., execution time or energy) through analysis of instrumentation data from execution of applications on real hardware. On average, the statistical models added to four scientific proxy applications resulted in simulations that are 12.5 times faster than the original application with only 6.08% error. This is on average 12.5% faster and 33.7% more accurate than baseline models.

Acknowledgment

This work was supported by the U.S. Department of Energy's National Nuclear Security Administration's Advanced Simulation and Computing program, the U.S. Department of Energy's Office of Advanced Scientific Computing Research, and Sandia National Laboratories' Laboratory Directed Research and Development program. Sandia National Laboratories is a multi-program laboratory managed and operated by Sandia Corporation, a wholly owned subsidiary of Lockheed Martin Corporation, for the U.S. Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

References

[1] J. J. Wilke, K. Sargsyan, J. P. Kenny, B. Debusschere, H. N. Najm, and G. Hendry, "Validation and uncertainty assessment of extreme-scale hpc simulation through bayesian inference," in *Proceedings of the 19th International Conference on Parallel Processing*, ser. Euro-Par'13. Berlin, Heidelberg: Springer-Verlag, 2013, pp. 41–52. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-40047-6_7

[2] Advanced Scientific Computing Advisory Committee (ASCAC) Subcommittee, "The opportunities and challenges of exascale computing," U.S. Department of Energy, Tech. Rep., 2010. [Online]. Available: http://science.energy.gov/~/media/ascr/ascac/pdf/reports/Exascale_subcommittee_report.pdf

[3] A. Kerr, E. Anger, G. Hendry, and S. Yalamanchili, "Eiger: A framework for the automated synthesis of statistical performance models," in *2012 19th International Conference on High Performance Computing*. IEEE, Dec. 2012, pp. 1–6. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6507525

[4] A. Rodrigues, R. Murphy, P. Kogge, J. Brockman, R. Brightwell, and K. Underwood, "Implications of a PIM Architectural Model for MPI," in *Proc. Cluster Computing*, 2003.

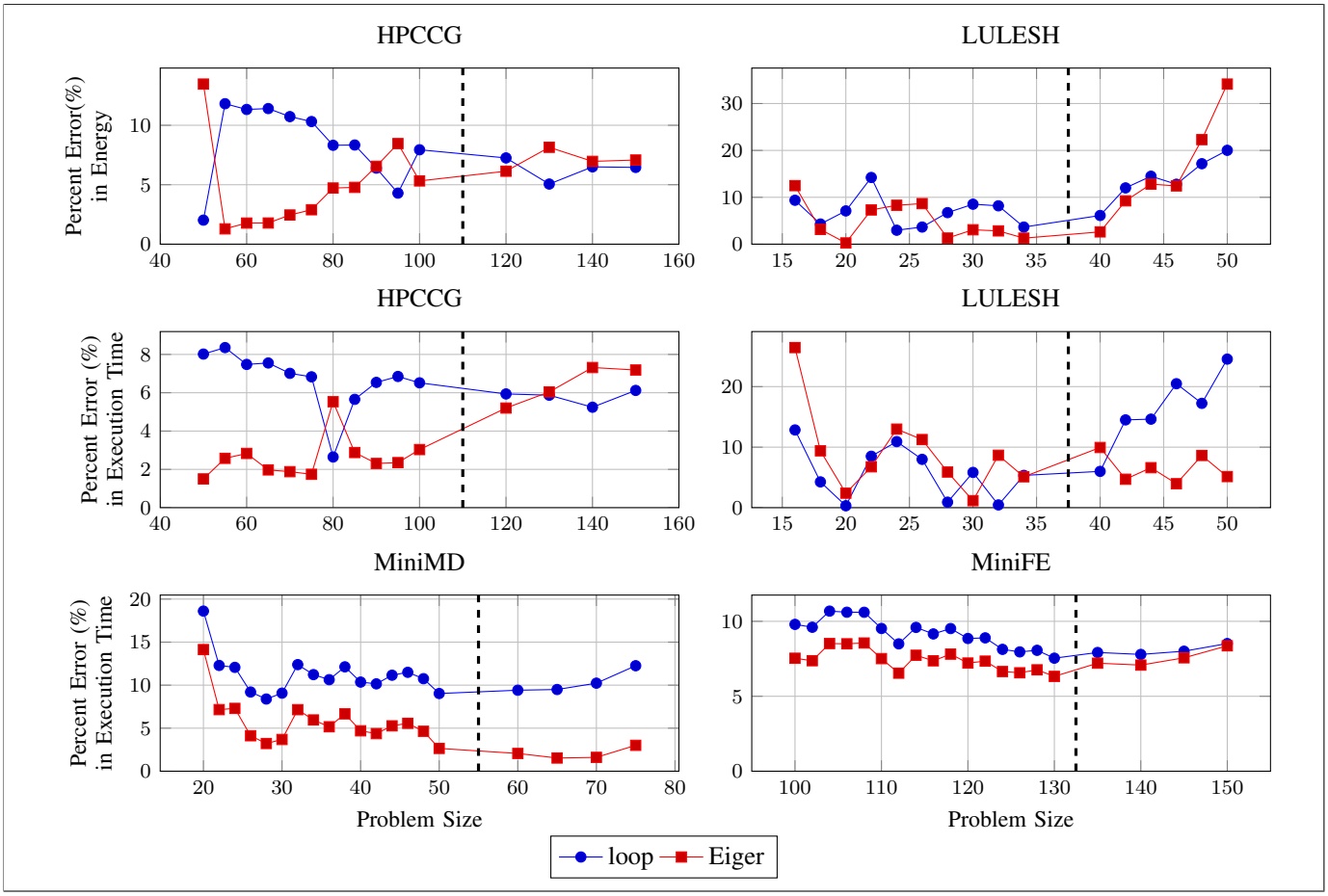

Fig. 8: Error in simulation predictions of application runtime.

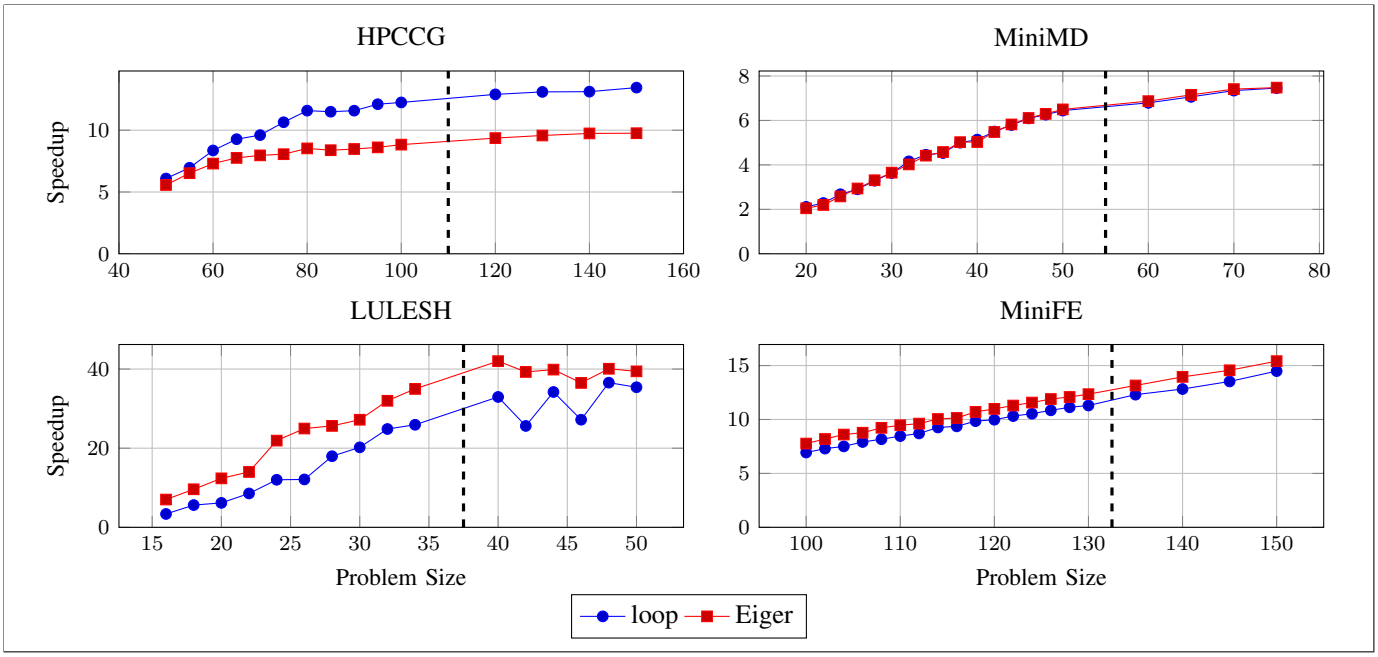

Fig. 9: Speedup of simulator runtime over native execution.

[5] K. D. Underwood, M. Levenhagen, and A. Rodrigues, "Simulating Red Storm: Challenges and Successes in Building a System Simulation," in *Proc. International Parallel and Distributed Processing Symposium (IPDPS'07)*. Los Alamitos, CA, USA: IEEE Computer Society, 2007, pp. 1–10.

[6] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. Kalé, "Simulation-based performance prediction for large parallel machines," *Int. Jour. Parallel Program.*, vol. 33, no. 2, pp. 183–207, June 2005.

[7] e. a. H Casanova, "SimGrid: A Generic Framework for Large-Scale Distributed Experiments," in *Iccms 2008: 10th Int. Conf. On Comput. Model. Simul.*, 2008, pp. 126–131.

[8] M. Tikir, M. Laurenzano, L. Carrington, and A. Snavely, "Psins: An open source event tracer and execution simulator for mpi applications," in *Euro-Par 2009 Parallel Processing*, ser. Lecture Notes in Computer Science, H. Sips, D. Epema, and H.-X. Lin, Eds. Springer Berlin / Heidelberg, 2009, vol. 5704, pp. 135–148.

[9] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob, "The structural simulation toolkit," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 37–42, March 2011.

[10] A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, and S. Pakin, "A Performance Comparison Through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/L, Red Storm, and Purple," in *SC '06: International Conference for High Performance Computing, Networking, Storage and Analysis*, 2006.

[11] S. Prakash, E. Deelman, and R. Bagrodia, "Asynchronous Parallel Simulation of Parallel Programs," *IEEE Transactions on Software Engineering*, vol. 26, no. 5, pp. 385–400, 2000.

[12] R. Riesen, "A Hybrid MPI Simulator," in *IEEE Inter. Conf. on Cluster Computing 2006*, sept. 2006, pp. 1–9.

[13] e. a. F Desprez, "Improving the Accuracy and Efficiency of Time-Independent Trace Replay," in *PMBS '12: 3rd Int. Workshop On Perform. Model. Benchmarking Simul. High Perform. Comput. Syst.*, 2012.

[14] V. Adve, R. Bagrodia, E. Deelman, and R. Sakellariou, "Compiler-optimized simulation of large-scale applications on high performance architectures," *Journal of Parallel and Distributed Computing*, vol. 62, no. 3, pp. 393–426, 2002.

[15] R. Susukita, H. Ando, M. Aoyagi, H. Honda, Y. Inadomi, K. Inoue, S. Ishizuki, Y. Kimura, H. Komatsu, M. Kurokawa, K. J. Murakami, H. Shibamura, S. Yamamura, and Y. Yu, "Performance prediction of large-scale parallell system and application using macro-level simulation," in *Proc. ACM/IEEE Conference on Supercomputing SC '08*. Piscataway, NJ, USA: IEEE Press, 2008, pp. 20:1–20:9.

[16] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, H. D. Simon, V. Venkatakrishnan, and S. K. Weeratunga, "The NAS parallel benchmarks–summary and preliminary results," in *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, ser. Supercomputing '91. New York, NY, USA: ACM, 1991, pp. 158–165. [Online]. Available: http://doi.acm.org/10.1145/125826.125925

[17] J. Dongarra, T. Hey, and E. Strohmaier, "PARKBENCH: Methodology, Relations and Results," in *HPCN Europe*, 1996, pp. 770–777.

[18] V. S. Adve, R. Bagrodia, J. C. Browne, E. Deelman, A. Dube, E. N. Houstis, J. R. Rice, R. Sakellariou, D. J. Sundaram-Stukel, P. J. Teller, and M. K. Vernon, "POEMS: End-to-End Performance Design of Large Parallel Adaptive Computational Systems," *IEEE Trans. Softw. Eng.*, vol. 26, no. 11, pp. 1027–1048, Nov. 2000. [Online]. Available: http://dx.doi.org/10.1109/32.881716

[19] J. Subhlok and Q. Xu, "Automatic construction of coordinated performance skeletons," in *Proceedings of the 2008 IEEE International Symposium on Parallel and Distributed Processing (IPDPS)*, April 2008, pp. 1–5.

[20] R. Preissl, T. Köckerbauer, M. Schulz, D. Kranzlmüller, B. R. d. Supinski, and D. J. Quinlan, "Detecting Patterns in MPI Communication Traces," in *Proceedings of the 2008 37th International Conference on Parallel Processing*, ser. ICPP '08. Washington, DC, USA: IEEE Computer Society, 2008, pp. 230–237. [Online]. Available: http://dx.doi.org/10.1109/ICPP.2008.71

[21] M. Sottile, A. Dakshinamurthy, G. Hendry, and D. Dechev, "Semi-Automatic Extraction of Software Skeletons for Benchmarking Large-Scale Parallel Applications," in *Proceedings of the 2013 ACM SIGSIM Conference on Principles of Advanced Discrete Simulation (ACM PADS)*, Montreal, Canada, May 2013.

[22] K. Hoste and L. Eeckhout, "Microarchitecture-Independent Workload Characterization," *IEEE Micro*, vol. 27, no. 3, pp. 63–72, May 2007. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4292057

[23] N. Goswami, R. Shankar, M. Joshi, and T. Li, "Exploring GPGPU workloads: Characterization methodology, analysis and microarchitecture evaluation implications," in *IEEE International Symposium on Workload Characterization (IISWC'10)*. IEEE, Dec. 2010, pp. 1–10. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5649549

[24] W. Jia, K. A. Shaw, and M. Martonosi, "Stargazer: Automated regression-based GPU design space exploration," in *2012 IEEE International Symposium on Performance Analysis of Systems & Software*. IEEE, Apr. 2012, pp. 2–13. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6189201

[25] S. A. M. Engin Ipek, Bronis R. De Supinski, Martin Schulz, "An approach to performance prediction for parallel applications," *Euro-Par*, 2005. [Online]. Available: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.85.2150

[26] A. Tiwari, M. A. Laurenzano, L. Carrington, and A. Snavely, "Modeling Power and Energy Usage of HPC Kernels," in *2012 IEEE 26th International Parallel and Distributed Processing Symposium Workshops & PhD Forum*. IEEE, May 2012, pp. 990–998. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=6270746

[27] B. Subramaniam and W.-c. Feng, "Statistical Power and Performance Modeling for Optimizing the Energy Efficiency of Scientific Computing," in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications & Int'l Conference on Cyber, Physical and Social Computing*. IEEE, Dec. 2010, pp. 139–146. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5724823

[28] C. L. Janssen, H. Adalsteinsson, S. Cranford, J. P. Kenny, A. Pinar, D. A. Evensky, and J. Mayo, "A simulator for large-scale parallel computer architectures." *IJDST*, vol. 1, no. 2, pp. 57–73, 2010.

[29] C. L. Janssen, H. Adalsteinsson, and J. P. Kenny, "Using simulation to design extremescale applications and architectures: programming model exploration," *SIGMETRICS Perform. Eval. Rev.*, vol. 38, pp. 4–8, March 2011.

[30] I. T. Jolliffe, *Principal Component Analysis*, 2nd ed. Springer Series in Statistics, 2002.

[31] ——, "A note on the use of principal components in regression," *Journal of the Royal Statistical Society. Series C (Applied Statistics)*, vol. 31, no. 3, pp. pp. 300–303, 1982. [Online]. Available: http://www.jstor.org/stable/2348005

[32] A. Kerr, G. Diamos, and S. Yalamanchili, "Modeling gpu-cpu workloads and systems," in *Third Workshop on General-Purpose Computation on Graphics Procesing Units*, Pittsburg, PA, USA, March 2010.

[33] M. H. Kutner, C. J. Nachtsheim, and J. Neter, *Applied Linear Regression Models*, fourth international ed. McGraw-Hill/Irwin, Sep. 2004.

[34] M. A. Heroux *et al.*, "Improving performance via mini-applications," Sandia National Labs, Tech. Rep. SAND2009-5574, September 2009. [Online]. Available: https://software.sandia.gov/mantevo

[35] S. Plimpton, "Fast parallel algorithms for short-range molecular dynamics," *J. Comput. Phys.*, vol. 117, no. 1, pp. 1–19, Mar. 1995, also at http://lammps.sandia.gov/index.html. [Online]. Available: http://dx.doi.org/10.1006/jcph.1995.1039

[36] J. Nickolls, I. Buck, M. Garland, and K. Skadron, "Scalable parallel programming with CUDA," *Queue*, vol. 6, no. 2, pp. 40–53, Mar. 2008. [Online]. Available: http://doi.acm.org/10.1145/1365490.1365500

[37] I. Karlin, J. Keasler, and R. Neely, "Lulesh 2.0 updates and changes," Lawrence Livermore National Laboratory, Tech. Rep. LLNL-TR-641973, August 2013.

[38] L. Dagum and R. Menon, "OpenMP: an industry standard API for shared-memory programming," *IEEE Computational Science Engineering*, vol. 5, no. 1, pp. 46–55, 1998.

[39] P. J. Mucci, S. Browne, C. Deane, and G. Ho, "Papi: A portable interface to hardware performance counters," in *In Proceedings of the Department of Defense HPCMP Users Group Conference*, 1999, pp. 7–10.