Performance Modeling for Systematic Performance Tuning

Torsten Hoefler
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
1205 W. Clark Street
Urbana, IL
htor@illinois.edu

William Gropp
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
1205 W. Clark Street
Urbana, IL
wgropp@illinois.edu

Marc Snir
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
1205 W. Clark Street
Urbana, IL
snir@illinois.edu

William Kramer
National Center for Supercomputing Applications
University of Illinois at Urbana-Champaign
1205 W. Clark Street
Urbana, IL
wkramer@ncsa.illinois.edu

ABSTRACT

The performance of parallel scientific applications depends on many factors which are determined by the execution environment and the parallel application. Especially on large parallel systems, it is too expensive to explore the solution space with series of experiments. Deriving analytical models for applications and platforms allow estimating and extrapolating their execution performance, bottlenecks, and the potential impact of optimization options. We propose to use such "performance modeling" techniques beginning from the application design process throughout the whole software development cycle and also during the lifetime of supercomputer systems. Such models help to guide supercomputer system design and re-engineering efforts to adopt applications to changing platforms and allow users to estimate costs to solve a particular problem. Models can often be built with the help of well-known performance profiling tools. We discuss how we successfully used modeling throughout the proposal, initial testing, and beginning deployment phase of the Blue Waters supercomputer system.

1. INTRODUCTION

The performance metric for national computing centers that support scientific users is the amount of "completed science per cost and time unit". For expensive systems, execution cost is a large, often dominant component of total cost. Optimizing for this metric can be done with several different techniques, such as

- application optimization (e.g., algorithmic optimizations as well as architecture-specific tuning)
- architecture optimization (e.g., selection of a suitable system architecture for the typical workload and workload-specific tuning)
- middleware optimization (e.g., optimized scheduling or topology mapping)
- policy optimizations (e.g., scheduling policy and priorities).

Those optimizations are often relative and specific to a particular installation and they can be done at the different stages in the lifetime of a computer system, i.e., early design and proposal, final preparation and planning, deployment, testing, and operation and maintenance. As computer systems require significant investments for a limited lifetime, it is important to utilize the systems efficiently from day one. Thus, application performance predictions and optimizations before the deployment phase gain importance.

In this paper, we present a holistic model-driven approach to tackle most optimization problems during all stages of system deployment and operation. Most important applications of a data-center's workload are ported from older systems without complete rewriting. We thus focus to maximize the performance of this workload of existing applications. Thus, the problem is to minimize the execution times of existing applications (or maximize the throughput) on a center's computing system.

The performance of parallel applications is complex. Complex system architectures, such as deep cache hierarchies, advanced out-of-order execution, or hierarchical interconnection networks present significant challenges to application performance analysis and prediction. Tuning applications for new systems is a significant investment, however, it is often unclear how certain changes affect the overall performance of an application. This is especially true for massively parallel systems or new architectures (e.g., accelerated computers).

Performance tuning is an activity performed by the algo-

SC'11 November 12-18, 2011, Seattle, Washington, USA
ACM 978-1-4503-0771-0/11/11.

rithm designers and application developers. Therefore, we focus on tools and methodologies that can be used by the application development teams, with limited interaction with performance specialists. We propose to use simple performance modeling to characterize the behavior of applications on computing systems. With this, we are able to choose between different optimization and architecture options.

In the following sections, we will present an overview of performance projection techniques, with a focus on analytic performance modeling. We also discuss the tradeoffs in accuracy and introduce a set of guidelines for modeling large application codes that we found useful in practice. We then present an example with a prominent and complex application and show how we use the models to guide optimizations and design.

2. PERFORMANCE PROJECTIONS

Projecting performance of a scientific application application can be done in multiple ways. Depending on the required accuracy and the available resources, one can choose between benchmarking the application on the target system, running a cycle-accurate simulation, a model-based (abstract) simulation, or analytic performance modeling.

Benchmarking.

is the most accurate performance projection. It takes all characteristics of the target architecture into account (it "runs the real code"). However, it provides very little insight and is mostly a black-box technique. Some architectures provide performance counters that can help to gain some insight. However, many architectural details are not accessible. Benchmarking also requires that the target architecture be available at the target scale, which is often not the case during the design phase and remains very expensive during the operational phase.

Detailed Simulation.

such as cycle-accurate CPU or network simulation [1, 2] can provide extremely accurate projections if the target hardware is not available. However, it often is thousands of times slower than the real execution and consumes significant memory resources [3]. Thus, very accurate simulation may not be an option for large scale systems. However, it can be very useful if it is applied to small kernels of applications to obtain execution times for these kernels that are then composed with an analytical model [4]. Simulation can provide much information into the application behavior, e.g., some CPU simulators allow investigation of each pipeline stage. However, the vast amount of data may not lead to the desired insight at the high level or require very high expertise or time investment to interpret the results correctly.

Model-based Simulation.

is a simulation technique where not each hardware detail is investigated but abstract models are used to determine the runtime or resource consumption. This technique forms the middle-ground between cycle-accurate simulation and analytical modeling. The simulation time can be less than the execution time leading to a "simulation speedup" of several orders of magnitude [4–7] while still accurately capturing important details of the execution. This technique has been used to discover important performance "effects" [8], however, it often fails to provide the required insight to understand the root cause of such effects. It is a complex task to execute such simulations in practice such that simulation tools are often only used by computer scientists and not by application developers. Nevertheless, model-based simulation is a very accurate practical and accurate technique to predict large-scale performance.

Analytical Performance Modeling.

is a technique where the performance is expressed with purely analytical expressions [9–11]. Such models need to be designed carefully to tradeoff the number of parameters versus the required accuracy. Models with fewer parameters are easier to generate and parameterize and provide more high-level insight while models with more parameters can be more accurate because they consider more system effects but can lead to overfitting. Abstract application performance models with a reasonably small number of parameters can be designed and maintained by the application developers while a system model can be provided by the system experts. Thus, we recommend this technique for all optimization purposes. Simpler models can be used as an interface to the application developers and algorithm designers while more complex models can be used for detailed tuning and projections.

For all described projection techniques, there exists a fundamental tradeoff between the number of parameters (e.g., details about the system architecture) and the accuracy of the model. Figure 1 provides a rough overview of this tradeoff and the different projection techniques. The high-level insight that can be gained from the model is also generally higher with less parameters, however, it is of course important that the prediction is characterizing the performance correctly.

Figure 1: Performance Projection Overview

In the following, we discuss analytic performance modeling techniques in detail followed by a discussion of the parameter-number to model-error tradeoff.

2.1 Analytic Performance Modeling

Analytic performance modeling represents application performance with analytic expressions. Those expressions can either model performance directly or indirectly. *Semi-empirical models* express the runtime of the application on a given architecture directly. *Application requirement models* are more flexible in that they express the application requirements, e.g., the needed number of floating point operations to solve the problem, independently of the architecture. Application requirement models can then be *instantiated* with a given *system model*, e.g., the floating point bandwidth, for a performance prediction.

We illustrate analytic performance models with the following simple matrix-matrix multiplication example:

```
for(int i=0; i<N; ++i)
  for(int j=0; j<N; ++j)
    for(int k=0; k<N; ++k)
      C[i+j*N] += A[i+k*N] * B[k+j*N];
```

Semi-Empirical Performance Modeling.

It is well known that the algorithm requires $\mathcal{O}(N^3)$ floating point operations to be solved. It is easy to see that the time needed to compute the triple-nested loop will also be of order N^3. We thus use the analytic model for the runtime $T(N) = t \cdot N^3$ to assess a semi-empirical model for the runtime. Semi-empirical modeling has only time as unit, thus t is specified in $[ns]$. Figure 2 shows a measurement series for different N on a POWER7 system and an empirically fitted model with $t = 2.2ns$ and an asymptotic standard error of 0.8%.

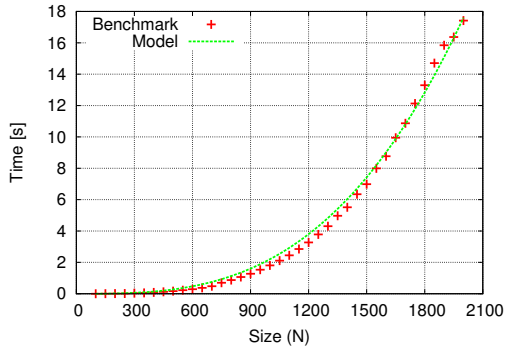

Figure 2: Semi-empirical Model for Matrix Multiplication on POWER7. The model shows $T(N)$.

We call this technique, where an analytic expression of the runtime is parameterized with empirical data *semi-empirical modeling.*

Application Requirement Modeling.

Instead of measuring the performance directly, we can measure the resource requirements of the problem to be solved (the *Application Requirement Model* or just *Application Model*).

The number of floating point operations required to solve the $N \times N$ matrix multiplication is $F(N) = 2 \cdot N^3$ (this was verified with performance counters). A second requirement, and potential bottleneck, are loads from main memory. The three loops are arranged to access stride-N and stride-1 doubles in A and B, respectively. Overall, the loop will still cause $\mathcal{O}(N^3)$ cache misses due to the stride-N access in the inner loop. However, $\mathcal{O}(N^2)$ of the previously fetched lines remain in cache and can be re-used for the stride-N accesses. Thus, we model the number of last-level cache misses as $C(N) = a \cdot N^3 - b \cdot N^2$. The unit of a and b are here $[cm]$ (cache misses).

Generating detailed application models for every memory access can be a complex task. Hence, we propose to use performance counters and curve fitting to determine the analytic application model. However, as opposed to semi-empirical modeling, the fitting is used to determine a parametrized analytic equation for the application requirements and not the overall performance. Figure 3 shows the measured L3 cache misses (measured with PAPI [12]) and the parametrized model function with $a = 0.00038cm$, $b = 0.278cm$, and an asymptotic standard error below 3%.

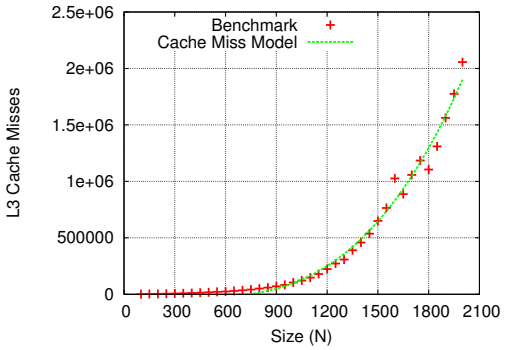

Figure 3: L3 Cache Miss Model for Matrix Multiplication on POWER7. The model shows $C(N)$.

Determining Bounds and Relative Performance.

Since the application requirements and the runtimes are known, one can now compute how well the functional units in the CPU are utilized. We investigate the floating point and memory rates for the matrix multiply example.

The floating point peak performance of the investigated POWER7 system is $R_{F,peak} = 3.864 \cdot 10^9 cycles/s \cdot 8fp/cycle = 30.912Gf/s$ if all instructions are vectorized and all multiply and add operations can be fused (FMA). Assuming that the floating point operations are the only bottleneck, the computation should scale with $T_{F,opt}(N) = 0.064ns \cdot N^3$. We remark that the inner loop has two floating point operations. However, the floating point performance of the simple implementation is $T(N) = 2.2ns \cdot N^3$ $(0.91Gf/s)$, which means that the performance is more than 33 times slower than the maximum.

One reason could be the that the performance is limited by the memory bandwidth of the system. The single-core memory read bandwidth of the system has been measured as $5GiB/s$. Each cache miss results in a 128 byte read request which results in an overall memory access time of $T_C = C(N) \cdot \frac{128b/cm}{5 \cdot 10^9 b/s} = 0.010ns \cdot N^3 - 7.12ns \cdot N^2$. Since $T_C < T_F$ and assuming overlap of computation and memory fetches in the out-of-order units, we conclude from our simple models that the code is bound by the floating point rate, however, it is far from optimal (33x). This is because the compiler fails to auto-vectorize and unroll the code. We will discuss model-driven optimizations for the simple code in Section 2.3.

We conclude that semi-empirical models can be used to express real application runtime while application requirement and system models can be used to characterize the quality of the implementation or the compilation tool-chain. Davidson used the latter technique to characterize the quality of the compiler [13]. The roofline model [14] is a related technique to assess the performance relative to the optimal performance. However, in our models, we also consider more potential application requirements, such as integer rate or memory latency depending on the application.

2.2 Accurracy Tradeoffs

We showed how to characterize the runtime of a matrix-matrix multiplication on a modern computing system with only a single parameter rather accurately. This semi-

empirical model is concisely representing a series of measurements with an error below 1% and allows extrapolation to larger matrix sizes that have not been measured.

In our proposed method, the model is specified *analytically* and instantiated *empirically*. This general method can be extended to application kernels where the application developer or performance engineer specifies a parametrized model which is then instantiated with measurements and curve fitting. The number of parameters in those models needs to be chosen to achieve the desired accuracy. For example, the constant loop startup overhead in the matrix matrix multiplication can be modeled with an additional parameter b, such that $T(N) = a \cdot N^3 + b$, however, b turned out to be insignificantly small in practice and was thus omitted.

Application models should be as simple as possible because this simplifies the design and parametrization and increases the insight. It might even be that the models cannot be used as accurate absolute performance predictors, however, they are merely enough to guide application or system design.

In fact, our simple analytic models are accurate enough to express performance concisely and guide optimization for complex real-world applications. We will discuss real-world examples in Section 4. In the next sections, we demonstrate how analytic models can be used during the different stages of system development and deployment.

2.3 Modeling for Performance Optimization

Providing help with optimizing the performance of parallel applications in particular computing systems is one of the main responsibilities of national computing centers. Experience shows that integrated approaches, where domain scientists (application programmers) and performance engineers (computer scientists) work together, are most successful [15].

2.3.1 *Identifying Optimization Opportunities*

Performance optimization strategies can roughly be split into two categories: (A) "tune until the performance is sufficient for my needs" and (B) "tune until the performance is within xx% of the optimum". Strategy (A) is most often used in practice, however, since supercomputer systems are large investments and a 10% speedup can often save millions of dollars (e.g., in the Blue Waters project [16]), strategy (B) can provide additional benefits. But it is often not easy to determine what the optimum performance is.

We use application requirements modeling, where we develop an application model for several parameters (e.g., required memory traffic, floating-point, or fixed-point instructions) and match it to the system model. This enables us to see if there are still optimization opportunities. For example, in the simple matrix multiplication as discussed before, we see that none of the system features is used fully and thus a huge optimization potential exists. In this case, the optimizations would require to use POWER7's Altivec instructions to achieve higher floating point rates. Figure 4 shows the runtime for an optimized matrix-matrix multiplication (ESSL 5.1's DGEMM) and the according semi-empirical model $T(N) = 0.072ns \cdot N^3$.

The number of required floating point operations in the application model is still $F(N) = 2N^3$, thus, the achieved floating point rate is $R_F = \frac{2fN^3}{0.072ns \cdot N^3} = 27.78Gf/s$ which is 90% of the peak floating point performance.

The memory bandwidth is still under-utilized. The optimized implementation has a lower I/O complexity than the unoptimized version: $C(N) = 0.00014 \cdot N^3 - 0.026 \cdot N^2$ with an asymptotic standard error of 2.5%, as shown in Figure 5, .

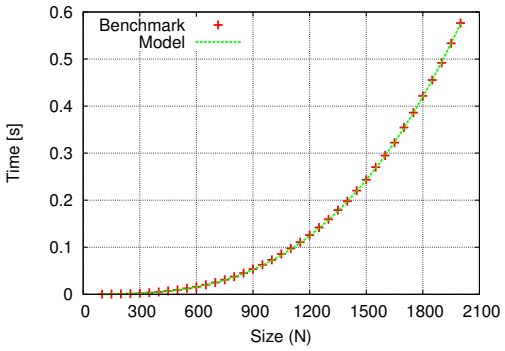

Figure 4: Execution time of optimized matrix multiplication on POWER7. The model shows $T(N)$.

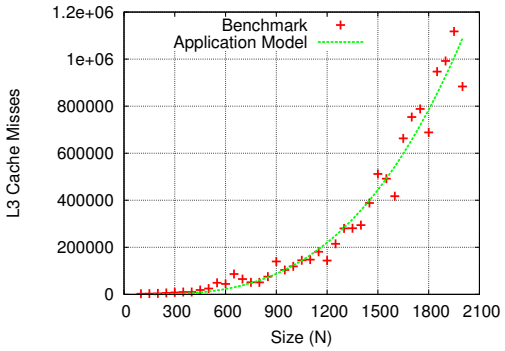

Figure 5: L3 Cache misses of optimized matrix multiplication on POWER7. The model shows $C(N)$.

2.3.2 *Choosing the Right Algorithm*

There is often a choice between different algorithms during the application implementation and optimization process. Decisions have to be made at all levels, for example, at the application level and at the implementation level. The application level could offer completely different algorithms for the solution of a scientific problem, such as real-space, Fourier space, or mixed computations in quantum molecular dynamics [17]. At the implementation level one has to make more detailed decisions, for example the order of loop nestings or cache blocking in the matrix matrix multiplication's simple or optimized implementation. Both schemes implement an algorithm with the same arithmetic complexity.

Implementing all different options is often very expensive and not feasible. Thus, we use simple models to classify the implementation options with regards to their relative performance. This means that we advocate the use of simple models as classifiers that do not necessarily predict the correct runtime but enable the selection of implementation options.

2.4 Modeling during Software Development

Performance modeling proved to be very useful in the optimization process of applications. We are experimenting with the integration of performance modeling into every step

of the software development process. In fact, this is often done implicitly by application developers, however, a structured approach and technique is missing. For example, in the well-known waterfall model, all steps can take advantage of modeling

Analysis Modeling is often used to pick the general method to be implemented. Rough asymptotic models are often used to pick the algorithm, for example, order N molecular dynamics [17].

Design In this phase, one identifies modules and functional units to be re-used. The algorithms and implementations for those models and functions are then selected with the help of simple (asymptotic) models. For example one would select quicksort ($\mathcal{O}(N \log(N))$) over bubblesort ($\mathcal{O}(N^2)$).

Implementation This phase deals with the details of the actual implementation. Performance modeling can be used to determine several implementation tradeoffs as shown with the matrix-matrix multiply example.

Testing Application testing should not only include correctness but also performance testing. The application performance over a larger number of experiments can often be expressed with a small set of parameters. Performance tests should monitor those parameters in regression analyses.

Maintenance Application performance can be used as an indicator of system health and to monitor any degradations in the system. Performance expectations can also be used to monitor soft errors or performance problems.

A useful strategy to utilize performance modeling during application development and operation would thus be to establish performance models during the design phase and validate and refine them through the whole application lifetime.

Now, we present a data-center-centric view of our performance modeling techniques.

2.5 Modeling during System Design and Proposal

We propose a method that can be used during system design (i.e., before the system is available). We model applications as sets of kernels (cf. Section 3.2). The performance of each of those serial kernels on a single CPU of the target architecture can be either be assessed empirically or analytically.

Empirical performance assessment can be performed by running each kernel on a single core of the target architecture or a single-core (or single-node) simulator. Analytic performance prediction requires an application model and can predict the maximum achievable performance (assuming ideal code and compiler). Application requirement models can be very helpful to decide on architectural system properties, such as the ratio between CPU floating point and memory access bandwidth or network bandwidth ratios.

The network can either be benchmarked at smaller scale with network measurement tools [18–22] or modeled analytically [23]. The application models will provide insight into the communication pattern and expected data sizes. We use different analytical methods and metrics to extrapolate from small networks to larger ones: bisection bandwidth, effective bisection bandwidth [24], and LogP [23] parameters.

We successfully applied this process during the proposal phase of the Blue Waters system. All application kernels were simulated using the Mambo POWER7 simulator [1]. Large-scale performance was predicted based on the expected network bandwidths and latencies and the bisection bandwidth of the PERCS topology [25]. Using this method, we were able to predict application performance on an architecture and system that was yet to be constructed.

2.6 Modeling during Deployment and Testing

The installation phase of a large computing system is often most important. Application models can be used to compare expected performance with measured performance as the system is brought up. This enables efficient functionality checks and the early detection of performance issues.

The models can also be used to demonstrate the final performance of the fully-deployed system. For example, in addition to a set of microbenchmarks to assess the final performance of the Blue Waters system, the contract with the vendor contains model-based performance estimates for several real-world applications that need to be reached to demonstrate the final full-system performance.

Thus, performance modeling can serve as an efficient tool for detecting and debugging performance issues as well as for establishing performance contracts between vendors and customers.

2.7 Modeling during Operation

When the system is deployed, the existing performance models can be used by the users (or even the system administrators) to monitor the operational performance of the system to detect a slow degradation or performance bugs due to failures.

If existing performance models can be formalized, then they could be used to automatically aid decisions during system operation. For example, batch schedulers could use the model information to predict the runtime of applications based on their input files and improve the scheduling efficiency. Schedulers could also base their node allocation on the model-based communication pattern of the application (e.g., densely communicating applications should be scheduled "closer together" than sparsely communicating applications). This could increase the efficiency of the overall system.

As discussed in Section 2.2, none of those models need to be ideal, in fact, even very rough application models, such as "heavy dense" vs. "light sparse" communication can aid the operational environment and increase performance.

In the next section, we will discuss a simple modeling strategy that can be used at different granularities (from very coarse to very detailed), depending on the required accuracy.

3. SIMPLE PERFORMANCE MODELING GUIDELINES

We propose a simple six-step process that can be applied to any existing application to model its performance. The first four steps are analytical, i.e., have to be determined from the source code or specified by a domain expert:

Step A1 Identify input parameters that influence runtime

Step A2 Identify application kernels

Step A3 Determine communication pattern

Step A4 Determine communication/computation overlap

The last two steps are empirical, i.e., are performed with series of benchmarks to instantiate the empirical or analytical performance models:

Step E1 Determine sequential baseline

Step E2 Determine communication parameters

The empirical steps can either be used to establish semi-empirical performance models, i.e., measure absolute performance, or analytical application requirement models, i.e., measure application requirements as described in Section 2.1.

3.1 Step A1: Identify all Input Parameters that Influence Runtime

This first step is most important as other steps are based on it. The task is to assemble a list of all input parameters that influence the runtime of the application. We call such input parameters *critical (input) parameters* in the following. Critical parameters should be scalar values such as sizes of dimensions or number of iterations. If the runtime is determined by an input file or a vector, then it should be condensed into the smallest number of scalar critical parameters (e.g., if the input file is a sparse matrix, the critical parameter could be the number of non-zero elements in the matrix). This step usually requires a domain expert to define the complete set of parameters.

It may not be simple to assess the impact of some critical parameters. For example, some parameters influence the number of iterations of a solver or load balancing strategies. Modeling their influence can be very complex and simplifying assumptions are often necessary (e.g., assume ideal load balance). When making simplifying assumptions, the modeler must be aware of the accuracy-simplicity tradeoffs that are introduced. If no good estimations can be made, then the model could leave some parameters as free parameter in the model (e.g., provide an estimate of the time per iteration but leave the number of iterations as parameter).

3.2 Step A2: Identify Code Kernels

The second step uses the list of critical parameters to assemble a list of functions or code blocks that are affected by them. Input-independent initialization functions can often be ignored while all other functions whose runtime depends on any critical parameter (even if it is negligible in small runs) should be listed. We refer to such input-dependent functions or code blocks as *kernels* in the following.

A **serial performance model** can either be an analytic application requirement model (e.g., counting the number of operations or loop iterations based on the critical parameters) or a semi-empirical performance model (e.g., benchmarking runtimes with different critical parameter settings, cf. Section 2.1). Instantiating analytic requirement-based models for a particular architecture can be less accurate than direct measurements due to the complexities in computer architectures (e.g., memory access patterns and caches). Empirical measurements, however, may need to cover the whole space of all critical parameters, which scales exponentially with the number of parameters. Thus, we propose to use empirical modeling, a combination of analytic modeling and empirical measurements. In semi-empirical modeling, one derives an analytic model and uses empirical measurements to parametrize it.

Analytic application requirement models can be constructed for each resource that an application demands, for example floating point operations, memory transactions, memory consumption. For example, a **memory consumption model** models the memory consumption of each kernel in the program. Often, this can be done by tracking all calls to memory allocation functions and determine their scaling with critical parameters. In rare cases, one would also need to investigate stack sizes and maximum recursion depth of function calls.

3.3 Step A3: Determine Communication Pattern

The next important application-specific step is to collect information about the communication pattern. Collective communications are typically simple to capture (only needs the type of collective operation [26]). However, it is important to also record information about the communicator size (it might be as simple as the total number of processes). For point-to-point communication patterns one needs to derive the logical pattern. For some applications, this can be derived from the source code (e.g., analyzing loops). The communication pattern of other applications may depend on the input file (e.g., semi-performing matrix-vector multiplications on sparse matrices). In this case, abstractions have to be introduced, e.g., the average number of neighbors and the average communication volume with regards to the critical input parameters. Models for those applications' communication requirements can be determined empirically. The resulting communication model should describe the data sizes and the communication structure.

It is often useful to express this in one of the well-known network models such as latency-bandwidth or LogGP.

3.4 Step A4: Determine Communication/-Computation Overlap

This step identifies all code regions where computation and communication can be overlapped. The modeler needs to extract the duration of the overlappable serial computation and communication for each kernel.

3.5 Step E1: Determine Sequential Baseline

Modeling of sequential performance is usually a very complex task. We propose to use a mixture of empirical and analytical modeling in that we define an analytical model and parametrize it with empirical measurements. This means that one designs a model for each kernel with a subset of the critical parameters as input. Then, one runs the code with a strategically chosen set of critical parameters and determines the time that each step takes. The analytic model is then fitted to the empirical measurements. Application requirement models can be constructed in the same way by measuring different values (e.g., cache misses for memory requirements).

Modeling cache effects and architecture details such as the superscalar reorder logic can be tricky, however, empirical measurements can avoid a complex analysis by capturing higher-order effects directly.

3.6 Step E2: Determine Communication Parameters

In order to parametrize a machine model, the communication parameters need to be determined. Ideally, all parameters (including collective communications) are specified by the vendor or are available in a central database. However, if this is not the case then the user can establish such models with the empirical modeling method. Several benchmarks are available to gather the parameters [18–22].

4. AN APPLICATION EXAMPLE: MILC

We applied the proposed technique to several applications. Due to space constraints, we will discuss an empirical performance model for one particular application on a POWER7 system in detail.

The discussed performance model is based on the model developed in [27, 28] and key features are repeated here to illustrate the modeling techniques described in Section 3.

The MIMD Lattice Computation (MILC) Collaboration studies Quantum Chromodynamics (QCD) the theory of the strong interaction [29]. Their suite of applications, known as the MILC code is publicly available for the study of lattice QCD. This group regularly gets one of the largest allocations of computer time at NSF supercomputer centers. The MILC suite comprises approximately 100.000 source lines of C code. We focus on one application from the code suite, su3_rmd, which is part of the SPEC CPU2006 and SPEC MPI benchmarks. It is also used to evaluate the performance of the Blue Waters computer based on a simple performance model prediction.

We show all steps to model MILC in full detail and present a complete analytical model for the parallel application performance. We omit step A4 because the application does not overlap computation and communication outside of the CG phase and the overlap for our desired problem size in the CG phase is insignificant.

4.1 Experimental Platform

We use a single POWER7-MR system with comparable single-core performance to predict the performance for larger scale (e.g., the Blue Waters system). The system is equipped with 32 POWER7 cores clocked at 3.864 GHz and 64 GiB main memory. We used the IBM XLC compiler version 11.1 and ESSL 5.1 for our experiments.

We used data gathered on this system to predict the performance of a POWER7-IH drawer (1024 cores total) running at 3.864 GHz. The main difference between POWER7-IH and POWER7-MR is the number of memory controllers, allowing more cores to achieve higher bandwidth, and the interconnect between the cores. However, single-core performance is comparable between both systems.

4.2 Step A1: Identify all Critical Parameters

MILC specifies a program run through an input file which contains all parameters. Table 1 lists relevant input parameters and describes their influence to the runtime briefly. Some variables, such as `nflavors1` or `nflavors2` are fixed by a particular type of calculation and are thus assumed to be part of the algorithm. The best way to gather all critical parameters is from the documentation or from a domain expert.

Name	Description
P	number of PEs (intrinsic parameter)
nx, ny, nz, nt	size in x, y, z, t dimension
warms, trajecs	warmup rounds and trajectories (outer loop)
traj_between_meas	measurement "frequency" (called `meas` for brevity)
steps_per_trajectory	number of "steps" in each trajectory (called `steps` for brevity)
beta, mass1, mass2, error_for_propagator	physics parameters that influence convergence of the conjugate gradient for measurements
max_cg_iterations	limits the conjugate gradient iterations

Table 1: MILC Critical Parameters.

4.3 Step A2: Identify all Code Kernels

The kernels are most important to our analysis. The runtimes of those blocks serve as the basis for the remaining model. We thus start with a serial performance model for all kernels. We use this model to compose a complete serial performance model for MILC and then extend it to a parallel performance model.

A kernel is either a function or a code block inside a function. A useful way to identify those kernels is to look at the call-graph of a representative run. Figure 6 shows such a callgraph for a run with a grid of size 4^4, one trajectory, and one measurement per trajectory.

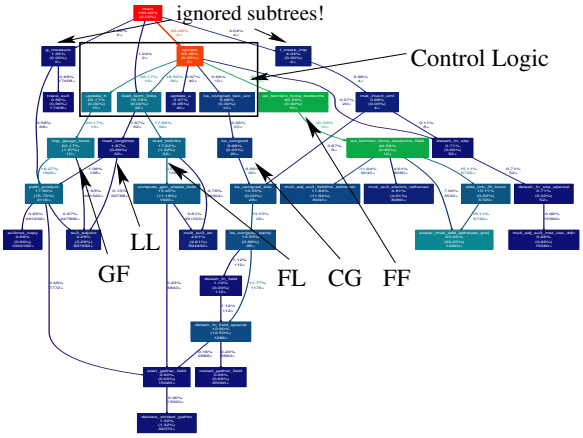

Figure 6: Callgraph for MILC, `trajecs=1`, `steps=5`, `meas=1`.

The main routine in the code loops over warmups and trajectories. Warmup rounds and trajectories are identical, however, warmup rounds don't produce output and don't perform "measurements". The bulk of the computation happens in the `update` routine which is called $2 \cdot$`trajecs` + `warms` times. This block is marked as "control logic" in Figure 6.

The code has five performance-critical kernels that account for most of the time: (1) LL (`load_longlinks`), (2) FL (`load_fatlinks`), (3) CG (`ks_congrad`), (4) GF (`imp_gauge_force`), and (5) FF (`eo_fermion_force_twoterms`). The time to perform each of these functions scales linearly with the number of lattice

sites. Let L_x, L_y, L_z, and L_t be the sizes of the dimensions of the lattice on each process and $V = L_x \cdot L_y \cdot L_z \cdot L_t$. Thus, the time to perform function $\mathcal{B} \in \{LL, FL, CG, GF, FF\}$ is $T(\mathcal{B}, V) = t(\mathcal{B}) \cdot V$ where $t(\mathcal{B})$ is the computation time of kernel $\mathcal{B}$ per grid point. The total serial runtime is $\sum_{b \in \mathcal{B}} T(b, V)$ assuming each kernel is executed once.

Choosing the right level of abstraction for modeling is very important. The modeler needs to decide where to cut subtrees in the call graph, i.e., combine them into a single term and he needs to determine functions to ignore. As shown in Figure 6, our modeling strategy ignores several functions in the callgraph (e.g., `trace_su3`). This was done after ensuring that those functions did not consume significant time **and** scaled asymptotically much slower (with all input parameters) than other modeled functions. This is the first of a number of tradeoffs to balance the complexity of the model with its accuracy.

The hardest part is to determine analytical models for the actual runtimes of the five kernels. We know that it scales linearly with the number of lattice sites per process, however, actual performance depends on memory layout, cache sizes, and architectural details of the CPU. Such a model is generally hard to predict and for this example, we choose simple semi-empirical performance modeling. We thus benchmark different local lattice sizes on the target system and match those to our model function $T(\mathcal{B}, V)$. Another possibility to determine this part of the model would be to run the kernels of the application in a simulator.

An analytic way would be to assess the requirements of the code, e.g., number of floating point operations and memory loads. The floating point requirements for the five identified kernels have been determined by the MILC group: $C(FF) = V \cdot 433,968F$, $C(GF) = V \cdot 153,004F$, $C(FL) = V \cdot 61,632F$, $C(LL) = 1,804F$, $C(CG) = V \cdot I \cdot 1,187F$ (I denotes the number of conjugate gradient iterations per trajectory). Such counts can give some approximation of the relative numbers of instructions for different parts of the code. We immediately see that LL and a single CG iteration have little demands compared to the other kernels.

However, the relationship between the number of FLOPs and real time is often non-trivial due to varying performance (vectorization, memory, cache) depending on the data layout, cache structure and CPU architecture.

4.4 Step E1: Determine Sequential Performance

Many modern computer systems have a memory hierarchy where each level can hold a different number of data elements and has different access times. This hierarchy is often transparent to the programmer in the form of one or more *caches* which keep temporary copies of parts of the application's working set in faster but smaller memory regions. Assessing the speed of a computation analytically can be very hard on cached architectures [30]. In addition, today's multi-scalar pipelined computer architectures make detailed analytical modeling of the execution units even more challenging. Thus, we rely on empirical benchmark results for different sizes of the application working set. Our analytical model considers two levels of cache: Lattice sites in the first level are computed in t_1 and sites in the second level are computed in t_2. The first level can hold $s(\mathcal{B})$ data elements.

$$T(\mathcal{B}, V) = t_1(\mathcal{B}) \cdot min\{s(\mathcal{B}), V\} + t_2(\mathcal{B}) \cdot max\{0, V - s(\mathcal{B})\} \quad (1)$$

We ran MILC with multiple different lattice sizes $V = L_x \cdot L_y \cdot L_z \cdot L_t$ in order to determine the single-core performance.

The following table provides the parameters for each critical block $\mathcal{B}$. It is important to notice that each block has different parameters which indicates that the working set and the time per site are different for the modeled operations.

The number of CG iterations depends on the complex critical parameters `beta`, `mass1`, `mass2`, `error_for_propagator` and can not easily be determined analytically and thus left as open parameter.

$\mathcal{B}$	$t_1(\mathcal{B})[\mu s]$	$t_2(\mathcal{B})[\mu s]$	$s(\mathcal{B})$
FF	62.4	92	3000
GF	27.8	48	4000
LL	0.425	0.68	4000
FL	11.4	20	3500
CG	0.239	-	∞

Figure 7 shows the performance model for the GF kernel (line), the actual benchmark results (crosses) and the relative error of the model for each measurement (stars at the bottom). The other four kernels show similar accuracy (less than 10% error).

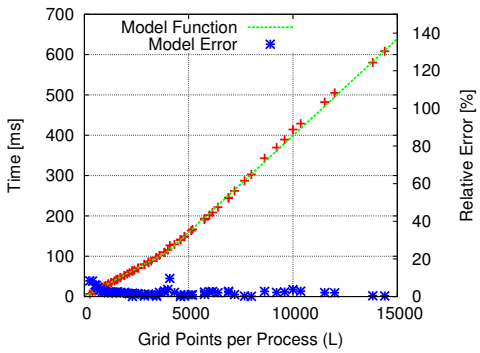

Figure 7: Emprical Performance Model for the GF Kernel in MILC. The model function is $T(GF, V)$, cf. Equation (1).

The serial performance of the MILC code for all combinations of critical input parameters can be concisely and precisely defined with 15 parameters. We now show how to construct an analytic equation that composes the single kernel runtimes.

4.4.1 *Putting it all Together*

In the serial case, L_x = `nx`, L_y = `ny`, L_z = `nz`, and L_t = `nt`, and $V = L_x \cdot L_y \cdot L_z \cdot L_t$. We determined the number of calls to each function depending on the critical parameters in Table 1 from the source code structure. The total serial computation time is

$$\begin{aligned} T_{serial}(V) \quad = \quad & (\texttt{trajecs} + \texttt{warms}) \cdot \texttt{steps} \cdot [T(FF, V) + \\ & T(GF, V) + 3(T(LL, V) + T(FL, V))] + \\ & \left\lfloor \frac{\texttt{trajecs}}{\texttt{meas}} \right\rfloor [T(LL, V) + T(FL, V)] + \\ & \texttt{niters} \cdot T(CG, V) \end{aligned} \quad (2)$$

The variable `niters` is the total number of conjugate gradient iterations for light and heavy quarks. The conjugate gradient method is called once per step and twice per measurement.

Figure 8 shows the composed serial performance model for the MILC code. The error is below 15% across a wide spectrum of configurations.

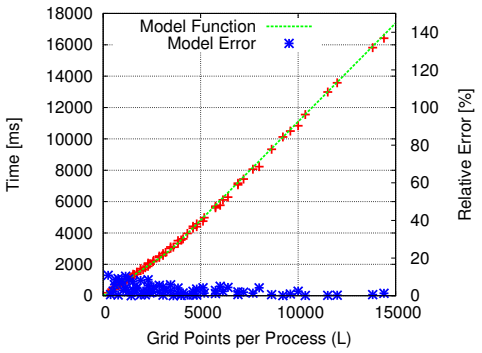

Figure 8: Composed Serial Performance Model for MILC. The model function is $T_{serial}(V)$, cf. Equation (2).

The number of iterations is different and hard to predict without domain knowledge, thus, we consider the total number of iterations in the model and resort to a domain expert to predict this number. We observed that beginning from a lattice size of 1536 points, `niters`≈23.000.

4.5 Step A3: Determine Communication Pattern

MILC uses point-to-point and collective (allreduce) communication. Collective communication is performed at the end of each conjugate gradient iteration on the whole set of processes (P).

4.5.1 Point-to-point Pattern

MILC uses a 4-d balanced domain decomposition scheme by trying to cut the largest dimension that is divisible by the largest prime-factor in P. It continues recursively after updating P (divide by prime factor) and the cut dimension. If no dimension can be cut by the largest prime factor, the program aborts.

In the following analysis, we assume that the domain is decomposed in all four dimensions and $L_x = L_y = L_z = L_t$. MILC models periodic boundary conditions such that all processes have exactly eight neighbors for $P \geq 16$. Point-to-point messages are sent along the 4-d lattice and are triggered in `gather` calls. Each gather call communicates in one direction and uses blocking communication, however, the conjugate gradient phase enables computation/communication overlap with nonblocking communication. The number of messages (gathers in MILC's terminology; this is not to be confused with MPI_Gather collectives!) $n(\mathcal{B})$ is specific to each critical block. In the case where all dimensions are cut by the domain decomposition:

$\mathcal{B}$	$n(\mathcal{B})$
FF	$(\texttt{trajecs} + \texttt{warms}) \cdot \texttt{steps} \cdot 1616$
GF	$(\texttt{trajecs} + \texttt{warms}) \cdot \texttt{steps} \cdot 828$
LL	$(3 \cdot \texttt{steps} \cdot (\texttt{trajecs} + \texttt{warms}) + \lfloor \frac{\texttt{trajecs}}{\texttt{meas}} \rfloor) \cdot 8$
FL	$(3 \cdot \texttt{steps} \cdot (\texttt{trajecs} + \texttt{warms}) + \lfloor \frac{\texttt{trajecs}}{\texttt{meas}} \rfloor) \cdot 288$
CG	$8 \cdot \texttt{niters} + 16 \cdot \texttt{nrestart} + 8 \cdot \lceil \texttt{steps} \cdot (\texttt{trajecs} + \texttt{warms}) + 2 \cdot \lfloor \frac{\texttt{trajecs}}{\texttt{meas}} \rfloor \rceil$

FF, GF, LL, and FL perform a fixed number of gathers per invocation. Each CG iteration performs one gather for each of the eight directions of all even (or odd) sites in a halo-zone of size three. In addition, at the first invocation (and each restart), it needs one additional gather for each direction (8 messages) of all even (or odd) sites in a halo zone of size one). For simplicity, we assume one restart during each invocation. Each CG invocation sends four messages communicating the `su3` matrices one poin deep and four messages communicating them three points deep. CG is invoked twice every step (light and heavy quarks) and four times every measurement.

Point-to-point Sizes.

The code uses two major types of point to point operations. The first type is used in FF, GF, LL, and FL and communicates `su3` matrices with 3x3 complex values (18 floating point values) and a 1-element wide halo zone. Thus, a halo zone of one element needs to be communicated at the domain boundaries. As before, L_x, L_y, L_z, and L_t represent the lattice dimensions per process, A_d represents the message size for the FF, GF, LL, and FL kernels along dimension d, and s is the size of a single floating point value:

$$A_x = 18 \cdot s \cdot L_y \cdot L_z \cdot L_t \qquad A_y = 18 \cdot s \cdot L_x \cdot L_z \cdot L_t$$
$$A_z = 18 \cdot s \cdot L_x \cdot L_y \cdot L_t \qquad A_t = 18 \cdot s \cdot L_x \cdot L_y \cdot L_z.$$

If we assume $L_x = L_y = L_z = L_t$ and $V = L_x \cdot L_y \cdot L_z \cdot L_t$, we get $A(V) = 18 \cdot s \cdot \sqrt[4]{V^3}$.

The CG kernel is more complex. It communicates either even or odd `su3` vectors with 3 element vectors (3 floating point values) per lattice site and a 3-element wide halo zone in each iteration. This means the message size is $B(V) = \frac{18}{2} \cdot s \cdot \sqrt[4]{V^3} = \frac{A(V)}{2}$. The conjugate gradient might restart itself to improve the accuracy of the solution. Each of the `nrestart` restart calls causes another 16 messages of size $\frac{B(V)}{3}$. We assume `nrestart`=1 for simplicity. We will write A instead of $A(V)$ and B instead of $B(V)$ where it is clear. In addition, each CG invocation sends four messages communicating the `su3` vectors with a halo-zone of size one (message size A) and four messages with a halo zone of size three (message size $3 \cdot A$). CG is invoked twice every step (light and heavy quarks) and four times every measurement.

Point-to-point Model.

If we assume a cost of $M(x)$ for a message of size x, the full communication model for point-to-point operations is:

$$\begin{aligned} T_{p2p} \quad = \quad & M(A) \cdot (\texttt{trajecs} + \texttt{warms}) \cdot \texttt{steps} \cdot (1616 + 828) + \\ & M(A) \cdot \left(3 \cdot \texttt{steps} \cdot (\texttt{trajecs} + \texttt{warms}) + \left\lfloor \frac{\texttt{trajecs}}{\texttt{meas}} \right\rfloor\right) \cdot \\ & (8 + 288) + 8M(B) \cdot \texttt{niters} + 16M\left(\frac{B}{3}\right) \texttt{nrestart} + \\ & (4M(A) + 4M(3A)) \cdot \\ & \left[\texttt{steps} \cdot (\texttt{trajecs} + \texttt{warms}) + 2 \cdot \left\lfloor \frac{\texttt{trajecs}}{\texttt{meas}} \right\rfloor\right] \end{aligned} \tag{3}$$

4.5.2 Collective Communication

Only the CG block requires an allreduce of two floating point numbers during each iteration. An additional allreduce call is needed for initialization at each first call for heavy or light quarks (once per step and twice for each measurement). Thus, the number of allreduce calls is:

$$n_{ared} = \texttt{niters} + 2 \cdot \left\lceil \texttt{steps} \cdot (\texttt{trajecs} + \texttt{warms}) + 2 \cdot \left\lfloor \frac{\texttt{trajecs}}{\texttt{meas}} \right\rfloor \right\rceil \quad (4)$$

The time for all collective operations is then simply $T_{coll} = T_{ared} \cdot n_{ared}$.

4.6 Step E2: Determine communication parameters

We used the Netgauge LogGP benchmark [19] to measure the intra-node communication performance and we assume the slowest link in the network (LR, cf. [25]) for the inter-node communication. The intra-node communication parameters are L=1.43 μs, for small messages ($\leq 32kiB$) o=0.48 μs, g=1.03 μs, G=0.18 ns/b, and for large messages ($> 32kiB$) o=5.66 μs, g=1.16 μs, G=0.22 ns/b. The intra-node parameters are L=1.8 μs, o=1.5 μs, g=1.5 μs, G=0.2 ns/b.

For 32 cores and an ideal mapping (two 2x2x2x2 blocks per node), we assume that approximately half of the communication is intra-node and half is inter-node.

We estimate the costs of the allreduce communication as a dissemination pattern [31]: $T_{ared} = 1.8\mu s \cdot \log_2(P)$.

5. EXAMPLE USES OF THE MODELS

We now discuss two examples in which we used the model to predict the performance on a system during installation and to assess the optimization potential of a code change.

5.1 POWER7-IH Prediction

We used the model to predict the performance of a parallel execution of MILC with $P = 1024$ and varying grid sizes. The simple model we used was $T_{par}(V) = T_{serial}(V) + T_{p2p}(V) + T_{coll}(V)$. Figure 9 shows the parallel model (line), the benchmark results (stars), the purely serial (computation) model (dashed line), the relative communication overhead (line from the left top), and the relative overhead for packing data for the point-to-point communication (lower line from bottom left).

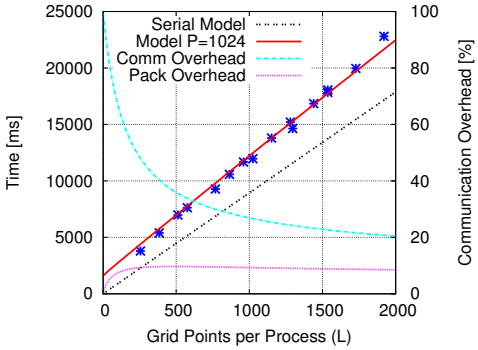

Figure 9: Parallel Performance Model for MILC. The model function is $T_{par}(V)$.

The "pack time" is a part of the communication overhead. It models the time to copy the data that is sent from the application buffer to communication buffers and is modeled with a simple linear equation.

5.2 MPI Datatype Optimization

Our communication model showed that the buffer copies for packing were very inefficient (only 300 MiB/s transfer rates). The model shows up to 12% overhead due to packing. Hoefler and Gottlieb thus changed the code to use optimized MPI derived datatypes for the data packing and sending (which also allows overlap and pipelining inside the send call). Details of the implementation are available in [32].

The resulting code's overall parallel performance was improved by up to 12% on the POWER7 drawer, which means that all the memory copy overhead could be eliminated in the efficient datatype handling.

6. CONCLUSIONS AND FUTURE WORK

We have shown that performance modeling can support all stages of the procurement and deployment of a large computer system and improve the day-to-day operation of datacenters. We also discussed how developers and users of scientific applications can benefit from a performance model. Therefore, we advocate performance modeling as a tool for application optimization and tuning, system design, procurement and tuning, and operation.

Performance modeling can be used at different levels of complexity, from very rough back-of-the-envelope to very detailed and accurate models. We advocate simple models in which we trade off simplicity for accuracy to guide optimization and tuning. Our simple and effective modeling method can be used by performance engineers, application support engineers, and application developers to generate performance models for a scientific applications without the help of experts. We provide a detailed example for MILC, an important NSF application and showed two practical uses of the model at the National Center of Supercomputing Applications (NCSA).

NCSA is developing a long-term strategy to support performance modeling and use performance models during its operation. The advanced application support team is well-versed with modeling strategies and begins to support external users with the development of performance models. We expect that this support strategy will enhance the quality and performance of many community codes such as MILC.

We strive to make performance modeling techniques accessible to a wider community by showing the usefulness of simple performance models. We also work with performance tool developers to integrate performance modeling techniques in such tools [33].

Our current ongoing work is using analytic performance models to predict the impact of operating system noise at large scale [8] and to develop noise-resistant algorithms using nonblocking collective communications. We also plan to investigate the limits to the benefit of accelerators. We plan to build upon asymptotic theoretical bounds on the I/O established in [34].

Acknowledgments

The authors thank Steven Gottlieb of Indiana University, who spent a 2009–2010 sabbatical at NCSA in order to model and tune performance of the MILC code for Blue Waters. The authors thank Greg Bauer and Robert Fiedler at NCSA for discussions and useful comments. This work is supported by the Blue Waters sustained-petascale computing project, which is supported by the National Science Foundation (award number OCI 07-25070) and the state of Illinois.

7. REFERENCES

[1] Patrick Bohrer, James Peterson, Mootaz Elnozahy, Ram Rajamony, Ahmed Gheith, Ron Rockhold, Charles Lefurgy, Hazim Shafi, Tarun Nakra, Rick Simpson, Evan Speight, Kartik Sudeep, Eric Van Hensbergen, and Lixin Zhang. Mambo: a full system simulator for the powerpc architecture. *SIGMETRICS Perform. Eval. Rev.*, 31:8–12, March 2004.

[2] Arun F Rodrigues, Richard C Murphy, Peter Kogge, and Keith D Underwood. The structural simulation toolkit: exploring novel architectures. In *Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, SC '06, New York, NY, USA, 2006. ACM.

[3] Chao Mei. A preliminary investigation of emulating applications that use petabytes of memory on petascale machines. Master's thesis, University of Illinois at Urbana-Champaign, 2007.

[4] Gengbin Zheng, Gagan Gupta, Eric Bohm, Isaac Dooley, and Laxmikant V. Kale. Simulating Large Scale Parallel Applications using Statistical Models for Sequential Execution Blocks. In *Proceedings of the 16th International Conference on Parallel and Distributed Systems (ICPADS 2010)*, number 10-15, Shanghai, China, December 2010.

[5] Rosa M Badia, Jess Labarta, and Judit Gimenez. Dimemas: Predicting mpi applications behavior in grid environments. In *Workshop on Grid Applications and Programming Tools (GGF '03)*, 2003.

[6] Mustafa M. Tikir, Michael A. Laurenzano, Laura Carrington, and Allan Snavely. PSINS: An Open Source Event Tracer and Execution Simulator for MPI Applications. In *Euro-Par '09*, pages 135–148, Berlin, Heidelberg, 2009.

[7] Torsten Hoefler, Timo Schneider, and Andrew Lumsdaine. Loggopsim: simulating large-scale applications in the loggops model. In *Proceedings of the 19th ACM International Symposium on High Performance Distributed Computing*, HPDC '10, pages 597–604, New York, NY, USA, 2010. ACM.

[8] T. Hoefler, T. Schneider, and A. Lumsdaine. Characterizing the Influence of System Noise on Large-Scale Applications by Simulation. In *International Conference for High Performance Computing, Networking, Storage and Analysis (SC'10)*, Nov. 2010.

[9] T. Hoefler, R. Janisch, and W. Rehm. Parallel scaling of Teter's minimization for Ab Initio calculations. 11 2006. HPC Nano'06 in conjunction with the International Conference on High Performance Computing, Networking, Storage and Analysis, SC06.

[10] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Supercomputing '01: Proceedings of the 2001 ACM/IEEE conference on Supercomputing (CDROM)*, pages 37–37, New York, NY, USA, 2001. ACM.

[11] Mark M. Mathis, Nancy M. Amato, and Marvin L. Adams. A general performance model for parallel sweeps on orthogonal grids for particle transport calculations. Technical report, College Station, TX, USA, 2000.

[12] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, Supercomputing '00, Washington, DC, USA, 2000. IEEE Computer Society.

[13] Gheith A. Abandah and Edward S. Davidson. Modeling the Communication Performance of the IBM SP2. *Parallel Processing Symposium, International*, 0:249, 1996.

[14] Samuel Williams, Andrew Waterman, and David Patterson. Roofline: an insightful visual performance model for multicore architectures. *Commun. ACM*, 52:65–76, April 2009.

[15] Abhinav Bhatele, Sameer Kumar, Chao Mei, James C. Phillips, Gengbin Zheng, and Laxmikant V. Kale. NAMD: A Portable and Highly Scalable Program for Biomolecular Simulations. Technical Report UIUCDCS-R-2009-3034, Department of Computer Science, University of Illinois at Urbana-Champaign, February 2009.

[16] Blue Waters Sustained Petascale Computing, Project Office. http://www.ncsa.illinois.edu/BlueWaters/, 2011. accessed June 2011.

[17] A. Canning, G. Galli, F. Mauri, A. De Vita, and R. Car. O(n) tight-binding molecular dynamics on massively parallel computers: an orbital decomposition approach. *Computer Physics Communications*, 94(2-3):89 – 102, 1996.

[18] T. Hoefler, T. Mehlan, A. Lumsdaine, and W. Rehm. Netgauge: A Network Performance Measurement Framework. In *High Performance Computing and Communications, Third International Conference, HPCC 2007, Houston, USA, September 26-28, 2007, Proceedings*, volume 4782, pages 659–671. Springer, 9 2007.

[19] T. Hoefler, A. Lichei, and W. Rehm. Low-Overhead LogGP Parameter Assessment for Modern Interconnection Networks. In *Proceedings of the 21st IEEE International Parallel & Distributed Processing Symposium*. IEEE Computer Society, March 2007.

[20] Pallas GmbH. Pallas MPI Benchmarks - PMB, Part MPI-1. Technical report, 2000.

[21] Dave Turner, Adam Oline, Xuehua Chen, and Troy Benjegerdes. Integrating new capabilities into netpipe. In Jack Dongarra, Domenico Laforenza, and Salvatore Orlando, editors, *Recent Advances in Parallel Virtual Machine and Message Passing Interface,10th European PVM/MPI Users' Group Meeting, Venice, Italy, September 29 - October 2, 2003, Proceedings*, volume 2840 of *Lecture Notes in Computer Science*, pages 37–44. Springer, 2003.

[22] Ralf Reussner, Peter Sanders, and Jesper Larsson Träff. SKaMPI: A comprehensive benchmark for public benchmarking of MPI. *Scientific Programming*, 10(1):55–65, 2002.

[23] David Culler, Richard Karp, David Patterson, Abhijit Sahay, Klaus Erik Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: towards a realistic model of parallel computation. In *Principles Practice of Parallel Progr.*, pages 1–12,

1993.

[24] T. Hoefler, T. Schneider, and A. Lumsdaine. Multistage Switches are not Crossbars: Effects of Static Routing in High-Performance Networks. In *Proceedings of the 2008 IEEE International Conference on Cluster Computing*. IEEE Computer Society, Oct. 2008.

[25] B. Arimilli, R. Arimilli, V. Chung, S. Clark, W. Denzel, B. Drerup, T. Hoefler, J. Joyner, J. Lewis, J. Li, N. Ni, and R. Rajamony. The PERCS High-Performance Interconnect. In *Proceedings of 18th Symposium on High-Performance Interconnects (Hot Interconnects 2010)*. IEEE, Aug. 2010.

[26] T. Hoefler, W. Gropp, R. Thakur, and J. L. Traeff. Toward Performance Models of MPI Implementations for Understanding Application Scaling Issues. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 21–30. Springer, Sep. 2010.

[27] Greg Bauer, Steven Gottlieb, and Torsten Hoefler. Performance Modeling and Comparative Analysis of the MILC Lattice QCD Application su3_rmd. to appear.

[28] Steven Gottlieb. Personal communication about MILC code structure and main functions.

[29] Claude Bernard, Michael C. Ogilvie, Thomas A. DeGrand, Carleton E. DeTar, Steven A. Gottlieb, A. Krasnitz, R.L. Sugar, and D. Toussaint. Studying Quarks and Gluons On Mimd Parallel Computers. *International Journal of High Performance Computing Applications*, 5(4):61–70, 1991.

[30] A. Agarwal, J. Hennessy, and M. Horowitz. An analytical cache model. *ACM Trans. Comput. Syst.*, 7:184–215, May 1989.

[31] Debra Hengsen, Raphael Finkel, and Udi Manber. Two Algorithms for Barrier Synchronization. *Int. J. Parallel Program.*, 17(1):1–17, 1988.

[32] T. Hoefler and S. Gottlieb. Parallel Zero-Copy Algorithms for Fast Fourier Transform and Conjugate Gradient using MPI Datatypes. In *Recent Advances in the Message Passing Interface (EuroMPI'10)*, volume LNCS 6305, pages 132–141. Springer, Sep. 2010.

[33] T. Hoefler. Bridging Performance Analysis Tools and Analytic Performance Modeling for HPC. In *Proceedings of Workshop on Productivity and Performance (PROPER 2010)*. Springer, Dec. 2010.

[34] Hong Jia-Wei and H. T. Kung. I/o complexity: The red-blue pebble game. In *Proceedings of the thirteenth annual ACM symposium on Theory of computing*, STOC '81, pages 326–333, New York, NY, USA, 1981. ACM.