# Performance Engineering: A Must for Petaflops and Beyond

## Extended Abstract

Torsten Hoefler
University of Illinois at Urbana-Champaign
htor@illinois.edu

Marc Snir
University of Illinois at Urbana-Champaign
snir@illinois.edu

## ABSTRACT

We discuss the need for a more principled approach to the management of the performance of applications for petascale platforms and outline some initial successes, related to the Blue Waters project.

## Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Modeling techniques

## General Terms

Performance

## Keywords

High performance computing, Performance analysis, Performance modeling

## 1. MOTIVATION

Modern supercomputers are increasingly complex and expensive. For example, Blue Waters [6], a system with a peak performance of 10 Petaflop/s that is being installed at the University of Illinois, is funded by a $200M NSF grant; it will consume in excess of 10 Megawatts. Large computation may run for weeks, utilizing a large fraction of the system; they will cost millions of dollars. With such costs, it is imperative to run applications in the most efficient way possible. Indeed, while current resource allocation mechanisms may not encourage such a choice, it would be perfectly rational for a good expert to spend a year in order to improve performance by 10%.

Much of the work on application performance tuning is done today using a very ad-hoc trial and error process: Profiling is used to figure out where an application spends most of its time. Programmers then try various approaches in order to reduce the time consumed by these expensive modules. As the code is ported to a new platform, and as it is scaled to larger systems, the process is repeated. The process is repeated until the achieved performance is "good enough"; usually, the programmer does not know whether the achieved performance is close or far from the best achievable for the program on the target platform.

Various groups pursue interesting work on performance modeling tools and methodologies [5, 7] These can be used by experts to understand the performance of an application after the fact, or to predict its performance on a new system. These frameworks are not used by application development teams to guide their performance tuning work. *We believe that the use of performance modeling must be an integral part of application development and an essential tool for performance tuning.* This extended abstract motivates this belief and describes some reasonably successful approaches.


*LSAP'11,* June 8, 2011, San Jose, California, USA.
ACM 978-1-4503-0703-1/11/06.

## 2. METHODOLOGY

A good performance tuning methodology should achieve the following goals:

1. *Require limited experimentation at full size*: Such experiments are expensive; furthermore, because of the fast depreciation of computing equipment, it is important to be able to use a supercomputer efficiently as soon as it is deployed; much of the code development must occur before the platform is available.
2. *Be accessible to code developing teams:* It is not practical to develop large codes (many scientific codes today exceed 1 MLOCs), and then toss them over to the "performance experts". While the expertise in code performance may not be equally shared across all members of a team, we believe that each large HPC code development team must contain a "code performance expert".
3. *Indicates what code changes can improve performance and indicate the likely improvement to be obtained by any change.* Changing code is always tedious and error-prone; coding resources are always limited. It is important to be able to make informed choices about tuning strategies, and it is important to know what can be gained by further coding efforts.

Supercomputing applications are often run for very different input configurations, with varying number of processors. Only parametrized models can cover all this range – hence, a tuning strategy cannot be based entirely on measurements – it needs a performance model.

In many cases, the development team has an implicit performance model: For example, it understands whether it uses a linear or quadratic algorithm, it can estimate communication, etc. However, this model is seldom documented; furthermore, the translation from floating point operation counts or bytes transferred to running time is lacking.

This suggests a methodology that combines analytical modeling of applications – representing the knowledge of algorithm designers – together with empirical measurements of

the code or parts of it. The analytical model defines a formula for the performance of the code; the coefficients in this formula are estimated using measurements. Elements of Decision Theory can be applied to choose a set of measurements that minimizes uncertainty, for a given computation budget. For example, should one do more measurements at smaller scale, or fewer, at larger scale?

The *utilization* (aka *efficiency*) of a code is the ratio between its performance and the peak achievable performance. We often interpret "peak performance" as meaning the top number of floating point operations the system can perform, and compare this to the actual achieved floating point performance. This definition is misleading as floating point units are almost never the main performance bottleneck; communication, to memory and to other nodes, is much more likely to be the effective bottleneck.

This simple-minded definition of utilization can be extended to include consideration of additional resources: floating-point operations, memory bandwidth, message bandwidth per node, message count per node bisection bandwidth, etc. For each such resource, we have a measure of efficiency – the ratio between peak achievable and achieved. A combination of analysis and measurements can provide a model, not only for total compute time, but also for the number of memory accesses or number of messages communicated. Simple benchmarks can measure the peak capability for each resource (see, e.g., [8]).

Two factors complicate this calculus; first, one parameter may not be sufficient to characterize a resource; for example, memory bandwidth will be very different for sequential accesses and random accesses. The modeler will need to achieve a compromise between model simplicity and accuracy. For example, Snavely et al. assume that memory accesses fall in two categories: sequential and random [7]. Second, there can be complex interactions between the utilization for different resources. For example, better load balancing in an SMP node can lead to worse locality and worse memory traffic. A robust algorithm design should avoid regions where such phenomena result in brittle performance (e.g., by ensuring that tasks have sufficient granularity to amortize load balancing costs.) A robust architecture should minimize complex dependencies across the main resources.

In practice, it is often possible to split performance into three main components:

**core performance:** most heavily dependent on the memory subsystem and the instruction mix

**node performance:** focused on the interaction between cores (memory contention, local coordination, local load balancing, etc.)

**global communication** focused on the interation between nodes (network contention, global coordination, global load balancing, etc.)

# 3. APPLICATION EXAMPLE – MILC

The first author recently worked with Steven Gottlieb, one of the main developers of the MILC code [2], in preparation for its execution on Blue Waters.

First, the compute-intensive components of the code were identified by profiling; the analysis focused on them.

Based on discussions with Steven, the main parameters affecting performance were identified; an analytical model developed for operations per core in each of the code components. This, combined with actual measurements for different parameter values, provided a timing model for core computations.

A model was developed for global communications in each component: number of messages sent and their length (all communications are either point-to-point or collective allreductions). The actual communication time is estimated using a LogGP model [1] and using Netgauge to estimate the model parameters [4]. This step used information on the mapping of processes to actual nodes, in order to use different communication parameters for on-node and off-node message passing (the application uses one MPI process per core). A similar approach was used to estimate the cost of global operations.

Assuming no computation/communication overlap (a reasonable assumption for MILC), the sum of computation time and communication time yields en estimate for total execution time. This estimate needed to be adjusted, to reflect that the simultaneous execution on multiple cores results on memory congestion that slow down the cores. A simple fixed slow-down factor (10%, for a 16 way POWER5+ node) provided sufficient accuracy.

The approach was tested for a 120 node (120×16 core) POWER5+ system. The performance is predicted with an error of $\approx 1\%$.

The model has been used prescriptively to indicate bottlenecks; a 15% performance increase was achieved by replacing a packing loop with the use of MPI datatypes [3]

# 4. REFERENCES


[1] A. Alexandrov, M. Ionescu, K. Schauser, and C. Scheiman. LogGP: incorporating long messages into the LogP model. In *ACM symposium on Parallel algorithms and architectures*, pages 95–105. ACM, 1995.

[2] C. Bernard, C. DeTar, S. Gottlieb, U. Heller, J. Hetrick, L. Levkova, J. Osborn, D. Renner, R. Sugar, and D. Toussaint. Status of the MILC light pseudoscalar meson project. *Arxiv preprint arXiv:0710.1118*, 2007.

[3] T. Hoefler and S. Gottlieb. Parallel zero-copy algorithms for fast Fourier transform and conjugate gradient using MPI datatypes. *Recent Advances in the Message Passing Interface*, pages 132–141, 2010.

[4] T. Hoefler, A. Lichei, and W. Rehm. Low-overhead LogGP parameter assessment for modern interconnection networks. In *2007 IEEE International Parallel and Distributed Processing Symposium*, page 403. IEEE, 2007.

[5] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *2001 ACM/IEEE conference on Supercomputing*, pages 37–37. ACM, 2001.

[6] NCSA. The "blue waters" project. `http:\\www.ncsa.illinois.edu/BlueWaters/`.

[7] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *2001 ACM/IEEE conference on Supercomputing*. IEEE Computer Society, 2002.

[8] S. Williams, A. Waterman, and D. Patterson. Roofline: an insightful visual performance model for multicore architectures. *CACM*, 52:65–76, April 2009.