LA-UR-14-26577
Approved for public release; distribution is unlimited.

Title: Techniques for Automated Performance Analysis

Author(s): Marcus, Ryan C.

Intended for: Web

Issued: 2014-09-02 (rev.1)

Techniques for Automated Performance Analysis

Ryan Marcus*
Los Alamos National Laboratory, HPC-5
rmarcus@lanl.gov

LA-UR-14-26577

Abstract

The performance of a particular HPC code depends on a multitude of variables, including compiler selection, optimization flags, OpenMP pool size, file system load, memory usage, MPI configuration, etc. As a result of this complexity, current predictive models have limited applicability, especially at scale. We present a formulation of scientific codes, nodes, and clusters that reduces complex performance analysis to well-known mathematical techniques. Building accurate predictive models and enhancing our understanding of scientific codes at scale is an important step towards exascale computing.

1 Introduction

Increasing complexity in high performance computing is drastically increasing the number of variables that affect the performance of a scientific code, and exascale computing will only continue this trend [2]. As such, performance analysis and performance tuning are becoming increasingly difficult problems. In order to properly understand scientific codes, one needs a way to comprehend each variable and interaction involved. Previous works have focused on matching performance metrics to known hinderances [6, 8], identifying bottlenecks [4, 17], knowledge-generation systems [12], or genetic algorithms [15]. We focus on developing abstractions that reduce performance tuning and analysis problems to well-studied mathematical problems.

Traditionally, one may view a scientific code as a function that maps an input deck to a solution. For the purposes of performance analysis, we are more interested in the time-to-solution than the solution itself. We define a *compiler* C to be a function that maps some source code $s \in S$ and some compiler options $\{t_1 \in T_1, t_2 \in T_2, t_3 \in T_3, \dots\}$ to an executable $e \in E$.

*With substantial mentorship and contribution from Cornell Wright, David Gunter, and Dave Nystrom.

$$C : S \times T_1 \times T_2 \times T_3 \times \cdots \to E \tag{1}$$

We then define a *machine* M as a function that maps some executable $e \in E$, some input deck $i \in I$, and some execution parameters $\{p_1 \in P_1, p_2 \in P_2, p_3 \in P_3, \dots\}$ to a time-to-solution $t \in \mathbb{R}^+$.

$$M : E \times I \times P_1 \times P_2 \times P_3 \times \cdots \to \mathbb{R}^+ \tag{2}$$

We define the combination of some specific execution parameters and some specific compiler options as a *configuration* $x = \{p_1, p_2, \dots, t_1, t_2, \dots\}$, where $x_n = p_n$, and $x^n = t_n$. The domain of all possible configurations is called the *configuration space*, denoted by $\mathscr{F}$.

Note that the semantics of T_n and P_n are arbitrary: the value represented by T_i or P_i can be anything, as long as it is kept consistent. One could define P_1 as the number of MPI ranks and T_1 as the optimization level. For example, if one were to compile the source code of the `HYDRO` program using third-level optimizations with `GCC`, and then run the resulting executable on the `SUPER` cluster using 16 `MPI` ranks and the input deck `LARGE`, the time-to-solution could be represented by:

$$\texttt{SUPER}(\texttt{GCC}(\texttt{HYDRO}, 3), \texttt{LARGE}, 16)$$

For convenience, we also define an input deck generating function π that maps a source code $s \in S$ and a problem size $n \in \mathbb{Z}^+$ to an input deck $i \in I$, such that the input deck i represents a problem of size n capable of being solved by an executable generated from s.

$$\pi : S \times \mathbb{Z}^+ \to I \tag{3}$$

2 Strong and weak scaling analysis

Strong and weak scaling analysis are two of the most popular tools for performance analysis [14]. When comparing two different program configurations (compiler options and execution parameters), it is often beneficial to compare the strong and weak scaling exhibited by the two configurations.

We show that this kind of analysis can be automated and extended by formalizing strong and weak scaling, selecting a divergence metric, and searching for an optimal configuration. Further, we give examples about how this process provides insights into HPC codes.

2.1 Formalizing Strong and Weak Scaling

Strong scaling analysis is performed by varying the number of processors used to solve a problem of a constant size [14]. In practice, this is rarely done [10]. Ideally, increasing the number of processors by some factor $c \in \mathbb{Z}^+$ should decrease the time-to-solution by that

same factor. We can represent this ideal formally. For a given executable e generated from source code s and where P_1 represents the number of processors, we say that:

$$\frac{M(e, \pi(s,k), n, p_2, \dots)}{M(e, \pi(s,k), cn, p_2, \dots)} = c$$

Weak scaling analysis is performed by varying the number of processors used to solve a problem of proportional size [14], such that the amount of work *per-processor* is constant. Ideally, doubling the number of processors and doubling the problem size should not change the time-to-solution.

$$\frac{M(e, \pi(s,k), n, p_2, \dots)}{M(e, \pi(s,ck), cn, p_2, \dots)} = 1$$

Of course, these idealized ratios are almost never reached in practice. However, locating points where these ratios differ significantly from their ideal value can be useful in identifying scalability issues within applications. We can define two functions, α and β, which measure the difference between the observed ratio and the ideal ratio between n_1 processors and n_2 processors (the *scaling error between n_1 and n_2*) for strong and weak scaling, respectively.

$$\alpha_s^e(M, k, n_1, n_2, p_2, \dots) = \frac{n_2}{n_1} - \frac{M(e, \pi(s,k), n_1, p_2, \dots)}{M(e, \pi(s,k), n_2, p_2, \dots)} \tag{4}$$

$$\beta_s^e(M, k, n_1, n_2, p_2, \dots) = 1 - \frac{M(e, \pi(s,k), n_1, p_2, \dots)}{M(e, \pi(s, \frac{n_2}{n_1}k), n_2, p_2, \dots)} \tag{5}$$

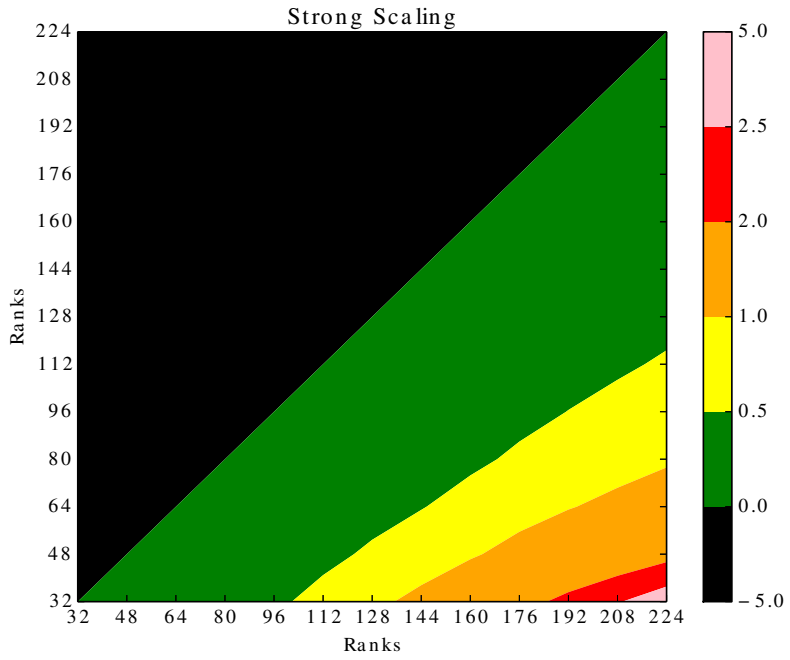
We define a *scaling* $\sigma \in \Sigma$ as a set of tuples (n_1, n_2, x) where x is the scaling error between n_1 and n_2 for some set problem size. This definition enables us to express strong and weak scaling as two functions, A and B, that give a scaling for a certain machine, compiler, configuration, and problem size. Formally, we can define A and B as:

$$\begin{aligned} A_k(M, C, s, p_2, p_3, \dots, t_1, t_2, \dots) = \{ \big(n_1, n_2, \alpha_s^{C(s,t_1,t_2,\dots)}(M, k, n_1, n_2, p_2, \dots)\big) \\ \mid n_1 \in \mathbb{Z}^+ \wedge n_2 \in \mathbb{Z}^+ \} \end{aligned} \tag{6}$$

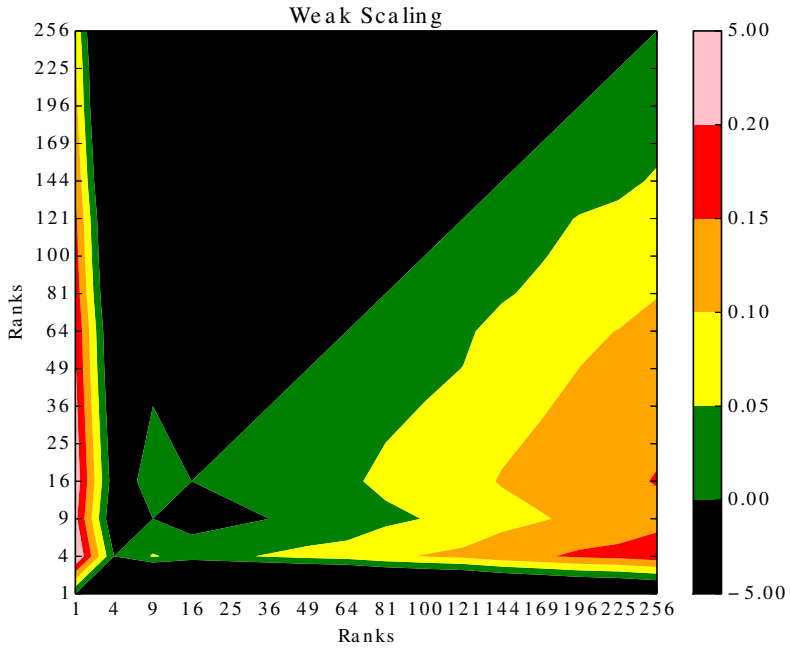
$$\begin{aligned} B_k(M, C, s, p_2, p_3, \dots, t_1, t_2, \dots) = \{ \big(n_1, n_2, \beta_s^{C(s,t_1,t_2,\dots)}(M, k, n_1, n_2, p_2, \dots)\big) \\ \mid n_1 \in \mathbb{Z}^+ \wedge n_2 \in \mathbb{Z}^+ \} \end{aligned} \tag{7}$$

By fixing the problem size (k), it is easy to plot both A and B. Plots of A and B[1] for a hydrodynamics code is shown in figure 1a and 1b, respectively. One can also plot a single row of the contours (ideally, the bottommost row), as shown in figures 2a and 2b.

[1] Obviously, we limit $\mathbb{Z}^+$ to a practical cardinality based on the number of nodes available.

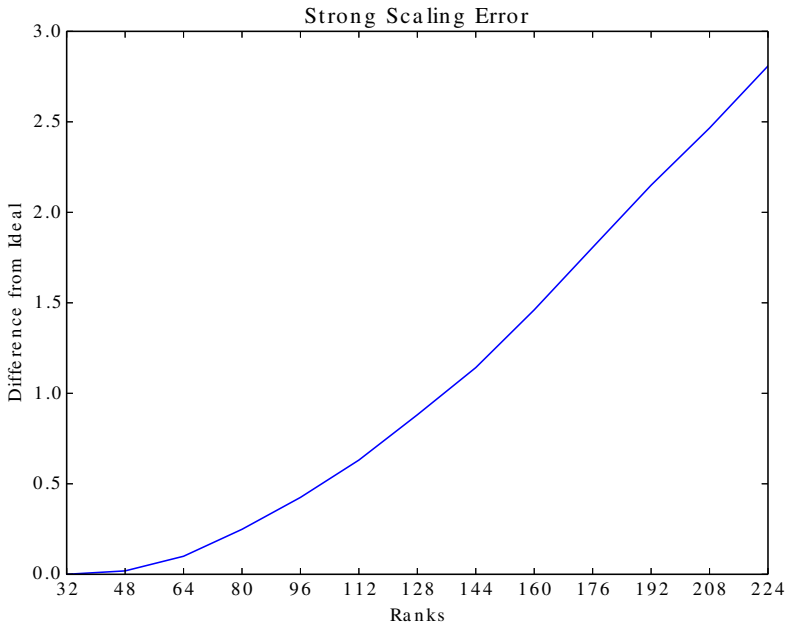


(a) Strong scaling error contour

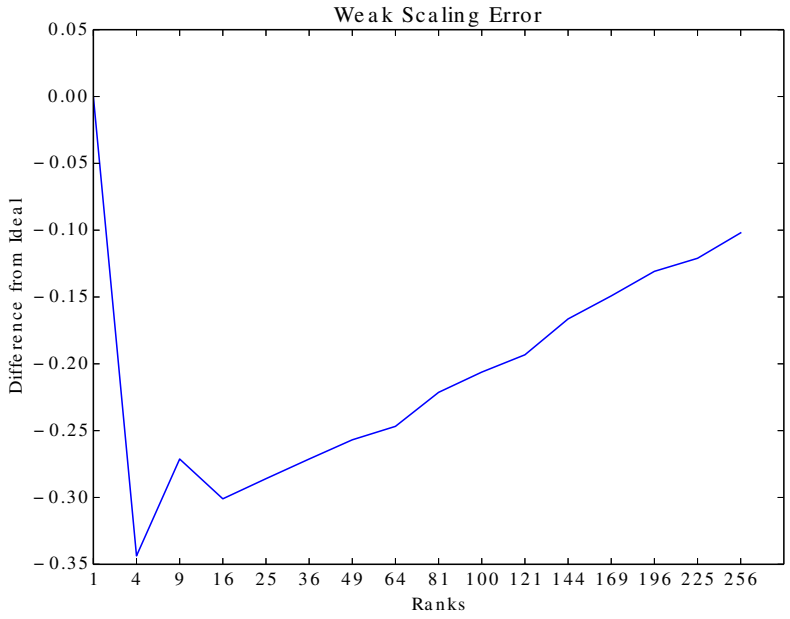


(b) Weak scaling error contour

Figure 1: Strong and weak scaling error contours



(a) Strong scaling error plot



(b) Weak scaling error plot

Figure 2: Strong and weak scaling error plots

2.2 Divergence Metric

From the shape and slopes of the plots in figure 2, a human can see that this particular code has exponential strong scaling error and fairly linear weak scaling error (starting at around 25 ranks). Given two strong scaling error plots, a human could also easily determine which plot depicted "better" scaling.

However, in order to programmatically determine which of two scaling plots has "better" scaling, one must develop a metric of how quickly error increases. We call this metric a *divergence metric*. We'll define a function $D : \Sigma \to \mathbb{R}$ such that $D(\sigma)$ is the divergence metric of some scaling σ. D needs to be defined in such a way that larger values of D represent a less desirable result than smaller values of D. Formally, for two scalings x and y, $D(x) < D(y)$ if and only if scaling x is preferable to scaling y.

One possible divergence metric is $-R$, where R is the correlation coefficient between n_2 and scaling error for a fixed n_1. Assuming that $\texttt{CorrCoef}(\delta)$ is the correlation coefficient of a set of 2-tuples ($z \in \delta \to z = (q \in \mathbb{R}, w \in \mathbb{R})$), we can define our divergence metric $D'_n(\sigma)$ as:

$$D'_n(\sigma) = -\texttt{CorrCoef}(\{(n_2, x) \mid (n, n_2, x) \in \sigma\})$$

This divergence metric has several drawbacks. D'_n will not always properly differentiate between linear error and exponential error. In fact, D'_n may label certain exponential errors as superior to certain linear errors, which is hardly ever the case. Pragmatically, we found D'_n to be a sufficiently capable divergence metric, but more complex codes may require more sophisticated metrics. This divergence metric would be especially poorly suited for codes that do not scale linearly (a linear increase in problem size does not create a linear increase in runtime).

2.3 Optimization

Mathematical optimization problems are classically stated as [3]:

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) < b_i \quad i = 1, \ldots, m \end{array}$$

We restate this slightly in order to allow for discrete constraints:

$$\begin{array}{ll} \text{minimize} & f_0(x) \\ \text{subject to} & f_i(x) \in b_i \quad i = 1, \ldots, m \end{array} \tag{8}$$

In equation 8, f_0 is called the *objective function*, and $f_1, \ldots, f_m$ are called the *constraint equations*. For our purposes, f_0 will be a mapping from configurations $x \in \mathscr{F}$ to divergences $d \in \mathbb{R}$ ($f_0 : \mathscr{F} \to \mathbb{R}$). We can define an f_0 for both weak and strong scaling, depending on which divergence we want to optimize. For strong scaling, f_0 is defined as such:

$$f_0(x) = D(A_k(M, C, s, x_2, x_3, \ldots, x^1, x^2, \ldots))$$

For weak scaling, f_0 is defined as:

$$f_0(x) = D(B_k(M, C, s, x_2, x_3, \ldots, x^1, x^2, \ldots))$$

Constraint equations can represent the constraints of a physical machine or compiler. For example, if T_1 is used to represent optimization level and one's compiler supports optimization levels 1, 2, and 3, one might let $f_1(x) = x^1 = t_1$ and $b_1 = \{1, 2, 3\}$. Constraint equations can also be used to eliminate non-optimal solutions. For example, if P_2 is used to represent the type of allocation (exclusive, shared, or other), one might let $f_2(x) = x_2 = p_2$ and $b_2 = \{\texttt{exclusive}\}$.

Having expressed the problem in terms of mathematical optimization, there are a large number of applicable techniques to find an optimal configuration [16]. In fact, many knowledge-based performance tuning techniques [6, 8, 12] are essentially complex heuristic functions utilized by standard optimization algorithms. Since the evaluation of A and B can be very costly, it is often advantageous to define enough constraints so that one's optimization algorithm only has to search in one or two dimensions at a time.

The result of optimization is not just an optimal member of $\mathscr{F}$, but also data about how configuration variables (p_n and t_n) affect performance. The optimal configuration itself is often less useful than insights gained from looking at which configuration variables produced the optimal configuration. While testing another HPC application, we found that the optimal configuration did not saturate each MPI slot with an MPI rank, suggesting that our application (at that problem size) was memory-bound, not CPU-bound. In the case of a certain hybrid MPI/OpenMP application, the optimal configuration used only one OpenMP thread. This revealed a workload distribution bug that had previously gone unnoticed.

3 Hotspot Analysis

Software profiling enables a user to see how much time specific functions in a program are using [9]. This sort of analysis is often called *hotspot analysis*, as it can reveal which functions are dominating a program's runtime [18]. However, in the world of HPC, the functions that dominate a program's runtime often change with the scale of the run. For example, at small sizes, an integration function might dominate, but at larger scales, communication functions [5] or decomposition algorithms may take longer.

We present a formulation of hotspots that enable standard mathematical techniques to gain insights into the runtime profile of HPC applications. First, we define some semantics of hotspots and hotspot collection. Then, we specify the requirements for a *distribution-difference function* and provide two examples. Finally, we explain how the output of such functions is useful for understanding HPC applications.

Hotspot Profile 1 (a)			Hotspot Profile 2 (b)		
Function	Proportion	Frequency	Function	Proportion	Frequency
function1	0.87	174	function1	0.82	328
function2	0.05	10	function2	0.08	32
function3	0.04	8	function3	0.05	20
function4	0.04	8	function4	0.05	20

Table 1: Two example hotspots from runs taking 200 seconds and 400 seconds. $\chi^2 = 2.684$ and $p = \mathscr{D}'(a, b) = 0.443$.

3.1 Formalizing Hotspots

Again, we use our formulations of compilers, source code, and executables, but we modify our formulation of a machine, M. Previously, the output of M was a time-to-solution. Now, we define M_H, which outputs a *hotspot profile*, which is a set of 2-tuples such that each tuple (x, y) contains the name of a function (x) and the proportion of a program's runtime that function took (y). Using $\mathbb{S}$ to represent the domain of all function names (strings), we have:

$$M_H : E \times I \times P_1 \times P_2 \times \cdots \to \{(x, y) \in (\mathbb{S}, \mathbb{R}^+)\} \tag{9}$$

We use $\mathscr{H}$ to refer to the domain of all hotspot profiles.

Assuming (again) that P_1 represents the number of processors used to run a problem, we define $H_s^C(M, k, n)$ to be the hotspot profile of running source s compiled by compiler C on machine M at a problem size k over n nodes with configuration $x \in \mathscr{F}$.

$$H_s^C(M, k, n, x) = M_H(C(s, x^1, x^2, \dots), \pi(s, k), n, x_2, x_3, \dots) \tag{10}$$

3.2 Distribution-difference function

As with the divergence function, a human could look at two different hotspot profiles and perhaps see if functions took more or less time as the scale changed. However, automating this process requires a distribution-difference function $\mathscr{D}$. Unlike the divergence function, this function maps two different hotspot profiles to a number in the range $\{0 \dots 1\}$, such that a number close to 0 means that the two profiles are very similar, while a number close to 1 means that the hotspot profiles are very different.

$$\mathscr{D} : \mathscr{H} \times \mathscr{H} \to \{0 \dots 1\} \tag{11}$$

We have a similar requirement for $\mathscr{D}$ as we did for D in that, for three hotspots $h_1, h_2, h_3 \in \mathscr{H}$, if h_1 is more similar to h_2 than it is to h_3, $\mathscr{D}(h_1, h_2) < \mathscr{D}(h_1, h_3)$.

One possible distribution-difference function is a Pearson's χ^2 test of independence, which measures the likelihood that differences in categorical data arose by chance [13].

Since we have proportional data between zero and one, we'll transform our proportions into time units by multiplying each value by the total runtime (in seconds). Consider each function as a category, and define the frequency of each category as the number of seconds spent in that function[2]. See Table 1 for an example.

With $a \in \mathscr{H}$ and $b \in \mathscr{H}$, let $|a|$ be the cardinality of a (the number of members of a). Then, in order to use the χ^2 test, we need to calculate the expected number of seconds for each function. E_a calculates the expected value for each function in the hotspot profile a and E_b calculates the expected value for each function in the hotspot profile b (where functions are indexed by i).

$$E_a(a,b,i) = \frac{(a_i + b_i) * \sum_n^{|a|} a_n}{\sum_j^{|a|}(a_j + b_j)}$$

$$E_b(a,b,i) = \frac{(a_i + b_i) * \sum_n^{|b|} b_n}{\sum_j^{|b|}(a_j + b_j)}$$

Then, we can use these two functions to create a distribution-difference function:

$$\mathscr{D}'(a,b) = \chi^2_{|a|-1}\left(\sum_i^{|a|} \frac{(a_i - E_a(a,b,i))^2}{E_a(a,b,i)} + \sum_i^{|b|} \frac{(b_i - E_b(a,b,i))^2}{E_b(a,b,i)}\right) \tag{12}$$

where χ^2_n is the right-tailed χ^2 probability with n degrees of freedom. Table 1 has example data with $\mathscr{D}'$ calculated.

3.2.1 Another possible distribution-difference function

Note that we can easily order the members of a hotspot profile based on the second member of each tuple (the proportion of time). In the event of a precise "tie", the order of the tied tuples may be arbitrary. Thus, if we consider two ordered arbitrary hotspot profiles $a \in \mathscr{H}$ and $b \in \mathscr{H}$, we can consider a_0 to be the largest value in a. Then, we can use a number of well-known rank comparison techniques [19] to determine if the ordering of a is substantially different from the ordering of b.

One such technique is Kendall Tau [11]. Given the following values:

- N, the total number of pairs, which is equal to $\frac{1}{2}|a|(|a|-1)$
- C, the total number of pairs whose ordering has not changed, i.e., if f_1 is before f_4 in a, and f_1 is also before f_4 in b.
- D, the total number of pairs whose ordering has changed, i.e., if f_2 is after f_1 in a, but f_2 is before f_1 in b.

[2]For exceptionally long runtimes, using minutes or even hours might be appropriate.

then we can compute τ as:

$$\tau = \frac{C - D}{N}$$

The value τ will be 1 if the two lists are perfectly similar, and -1 if the two lists are perfectly dissimilar. Thus, we can define a distribution-difference function as:

$$\mathscr{D}^T(a, b) = 1 - \frac{(1 + \tau)}{2} = \frac{1 - \tau}{2} \tag{13}$$

3.3 Analysis

One can use standard search techniques to find scales or problem sizes for which the hotspot profile changes substantially using either of the provided distribution-difference functions (equations 12 and 13), or other techniques. One can search for a point where the hotspot profile changes significantly (at a certain confidence), or one can plot the change in the hotspot profile as a problem scales. Regardless, one needs to define a function $f : \mathscr{F} \times \mathscr{F} \times \mathbb{Z}^+ \to \mathbb{R}$ that can be searched.

$$f(x, y, k) = \mathscr{D}(H_s^C(M, k, x_0, x), H_s^C(M, k, y_0, y))$$

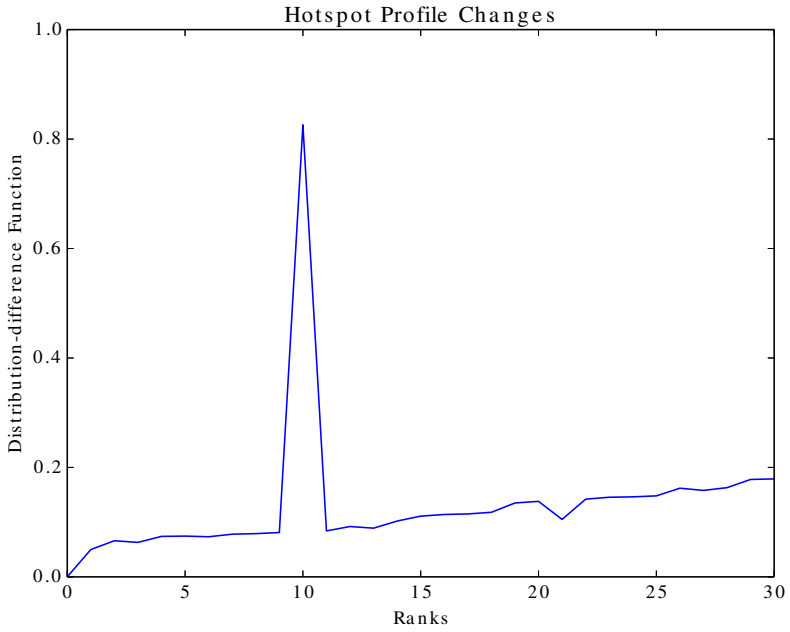
Notice that you can search in the configuration space (by varying x and y), or through various problem sizes (by varying k). Any number of searching techniques can be used, but a simple linear scan is normally a good starting point. Note that, while other performance analysis tools identify hotspots [17, 18, 9], the technique proposed here searches for differences in those identified hotspots.

This approach is especially useful for finding bugs in workload distribution functions. Often, when a problem is run on an odd (or prime) number of nodes using an odd (or prime) workload size, work is not distributed optimally, causing a large change in the hotspot profile. Figure 3a shows a large hotspot profile difference around rank 11, which revealed a workload distribution bug. Figure 3b shows the hotspot profile differences after fixing the bug.

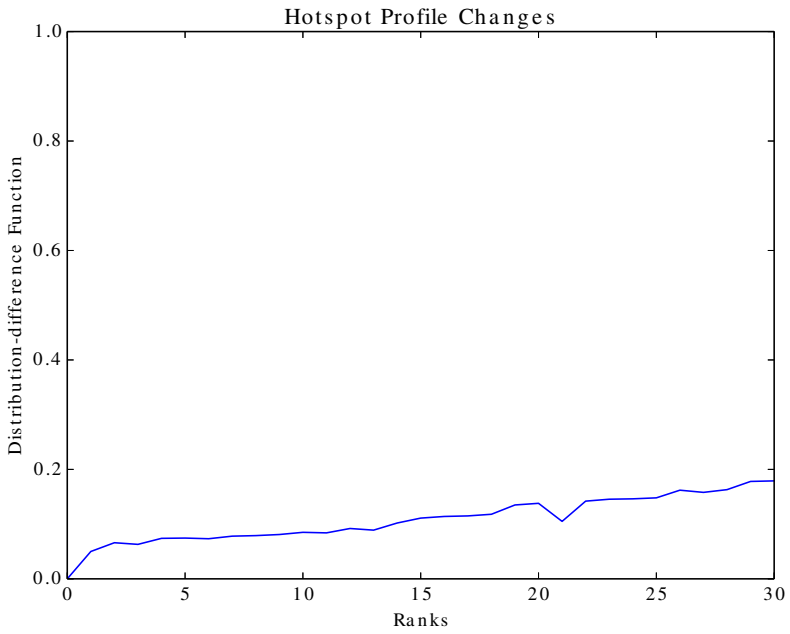
4 Machine-Machine Comparisons

Up until now, we have only looked at varying problem size and configuration. We have kept the machine, M, constant. However, the formalizations presented here can be extended to do some (basic) comparisons between two or more machines. This can be extremely useful when deciding how to allocate resources, and when deciding what additional computing capabilities improve performance.

We'll define a simple function that measures the performance of a code on a given machine. Then, we'll show how standard linear regression techniques can be used to compare the performance of two machines. Again, the nature of the formalizations will make it clear how the process can be automated.



(a) Before fixing workload distribution bug



(b) After fixing workload distribution bug

Figure 3: Hotspot profile differences using χ^2 distribution-difference function

4.1 Formalizing Machine-Machine Comparisons

We can use equation 2 to build two simple functions to measure the time-to-solution of a code $s \in S$ on two different machines, M and M'. Given a compiler C[3], a problem size k, and a configuration state x, the time-to-solution for M is:

$$M(C(s, x^1, x^2, \dots), \pi(s, k), x_1, x_2, \dots)$$

And for M', the time-to-solution is given by:

$$M'(C(s, x^1, x^2, \dots), \pi(s, k), x_1, x_2, \dots)$$

We then define a simple helper function, $h_m^{e,s,x}(k)$, which gives the time-to-solution of $s \in S$ running on machine m with configuration x, problem size k, and executable e:

$$h_m^{e,s,x}(k) = m(e, \pi(s, k), x_1, x_2, \dots) \tag{14}$$

4.2 Regression techniques

The standard form of a linear model [13] is:

$$y = a + bx$$

where a represents the y-intercept of the line and b represents the slope. For a given e, s, and x we can generate a number of samples on each machine and then use one of a number of techniques (like ordinary least squares) [1] to fit the data from each machine to a linear model.

First, select some subset $z \subset \mathbb{Z}^+$. Then, create a set of tuples $\hat{h}_m^{e,s,x}$ such that:

$$\hat{h}_m^{e,s,x} = \{(i, h_m^{e,s,x}(i)) \mid i \in z\} \tag{15}$$

We can then build a linear model of M using $\hat{h}_M^{e,s,x}$, and we can create a linear model of M' using $\hat{h}_{M'}^{e,s,x}$. This will give us two y-intercepts (a_M and $a_{M'}$) and two slopes (b_M and $b_{M'}$).

4.3 Analysis

The slopes of the generated equations are easy to interpret. If $b_M > b_{M'}$, then increasing the problem size on machine M will increase the time-to-solution more than the same increase in problem size on machine M'. Interpreting the intercepts can be slightly more

[3]Certain compilers may operate differently on different machines, or may only be available on specific platforms. Thus, comparing the performance of two machines often amounts to comparing the performance of two different hardware/software combinations. It may not always be possible to search or sample from the full domain of compilers or compiler options when comparing across two machines.

complicated. Generally, if one line is not above the other for all of $\mathbb{Z}^+$, there is something making one machine faster for smaller problem sizes. Perhaps one machine has more processors per node, a faster network interconnect, or a larger cache. Regardless, linear analysis provides a simple way to do a quick, automated comparison between two machines. It also enables one to quickly see how changes in a machine's hardware affect code performance (by comparing the linear models generated on the machine before and after the hardware change).

Using more complex regression techniques can provide different and often more useful insights into the performance signatures of two different machines. We show how to interpret a simple linear model, but since code performance is rarely perfectly linear [2], other models may prove more useful.

Until now, we have assumed that the input deck generation function (π) produces decks that have a particular performance signature that scales linearly (doubling the problem size approximately doubles the time-to-solution). In reality, this is rarely the case, as different input decks will often have different performance signatures. For example, one input deck may require a large amount of floating-point operations, while another input deck may require a large amount of memory. Using simple machine-machine comparisons can aid in matching an input deck's performance signature to a machine's capabilities, which can lead to improvements in an organization's workload distribution.

5 Conclusion

Several of the techniques presented here have helped to analyze and understand real-world HPC codes. These automated techniques greatly improve the quality of HPC models, and can aid developers in locating bugs and performance bottlenecks. While these techniques are not sufficient on their own, they make excellent additions to a programmer's performance analysis toolbox. Due to the automated nature of these techniques, they can be run frequently and unsupervised, making them good candidates for performance regression testing and analyzing multiple codes at once. With an increasing number of variables, exascale computing will require even more advanced techniques that can be applied in an automated manner.

A Mathematical Symbols and Notation Reference

$\mathbb{R}$	the domain of real numbers	$\mathbb{R}^+$	the domain of positive real numbers
$\mathbb{Z}$	the domain of integers	$\mathbb{Z}^+$	the domain of positive integers
$\mathbb{S}$	the domain of all strings	$\mathscr{H}$	the domain of all hotspot profiles
S	the domain of source code	E	the domain of executables
P_n	the domain of execution parameter n	T_n	the domain of compiler option n
$\mathscr{F}$	the domain of configurations	Σ	the domain of scalings
x_n	configuration notation for p_n	x^n	configuration notation for t_n
C	a compilation function (1)	M	a time-to-solution (machine) function (2)
π	the input deck generation function (3)	σ	a scaling
α	strong scaling error function (4)	β	weak scaling error function (5)
A	strong scaling generator function (6)	B	weak scaling generator function (7)
D	divergence metric	$\mathscr{D}$	distribution difference function (11)
M_H	hotspot profile generator function (9)	H_s^C	hotspot profile for s compiled by C (10)
$h_m^{e,s,x}$	time-to-solution helper function (14)	$\hat{h}_m^{e,s,x}$	tuples of points sampled from $h_m^{e,s,x}$ (15)

We use set builder notation [7] to construct sets.

$$p = \{x \mid F(x) \wedge G(x)\} \tag{16}$$

The notation in equation 16 can be read as "the set of all x such that $F(x)$ and $G(x)$ is true." In other words, p is the set of any and all possible values of x such that $F(x)$ and $G(x)$ are true.

Equation 16 can be stated equivalently as:

$$(x \in p) \rightarrow [F(x) \wedge G(x)]$$

Or as:

$$\forall x(F(x) \wedge G(x)) \quad x \in p$$

References

[1] Michael Patrick Allen. *Understanding regression analysis.* Springer, 1997.

[2] Steve Ashby, Pete Beckman, Jackie Chen, Phil Colella, Bill Collins, Dona Crawford, Jack Dongarra, Doug Kothe, Rusty Lusk, Paul Messina, and Others. The opportunities and challenges of exascale computing. *summary report of the advanced scientific computing advisory committee (ASCAC) subcommittee at the US Department of Energy Office of Science*, 2010.

[3] Stephen Boyd and Lieven Vandenberghe. *Convex Optimization.* Cambridge University Press, New York, NY, USA, 2004.

[4] Harold W. Cain and et al. A callgraph-based search strategy for automated performance diagnosis, 2000.

[5] Laura C Carrington, Michael Laurenzano, Allan Snavely, Roy L Campbell, and Larry P Davis. How well can simple metrics represent the performance of hpc applications? In *Supercomputing, 2005. Proceedings of the ACM/IEEE SC 2005 Conference*, pages 48–48. IEEE, 2005.

[6] Guojing Cong, I-Hsin Chung, Hui-Fang Wen, D. Klepacki, H. Murata, Y. Negishi, and T. Moriyama. A systematic approach toward automated performance analysis and tuning. *Parallel and Distributed Systems, IEEE Transactions on*, 23(3):426–435, March 2012.

[7] H.B. Enderton. *A Mathematical Introduction to Logic.* Harcourt/Academic Press, 2001.

[8] Michael Gerndt and Andreas Krumme. A rule-based approach for automatic bottleneck detection in programs on shared virtual memory systems. In *High-Level Programming Models and Supportive Environments, 1997. Proceedings., Second International Workshop on*, pages 93–101. IEEE, 1997.

[9] Susan L. Graham, Peter B. Kessler, and Marshall K. Mckusick. Gprof: A call graph execution profiler. *SIGPLAN Not.*, 17(6):120–126, June 1982.

[10] John L Gustafson. Reevaluating amdahl's law. *Communications of the ACM*, 31(5):532–533, 1988.

[11] Michiel Hazewinkel, editor. *Encyclopaedia of Mathematics.* Kluwer Academic Publishers, 2 2002.

[12] B Robert Helm, Allen D Malony, and SP Fickas. Capturing and automating performance diagnosis: the poirot approach. In *Parallel Processing Symposium, 1995. Proceedings., 9th International*, pages 606–613. IEEE, 1995.

[13] Barbara Illowsky. *Introductory statistics*. OpenStax College, Rice University, Houston, Texas, 2013.

[14] Alan Kaminsky. *BIG CPU, BIG DATA*. Rochester Institute of Technology - Department of Computer Science, online edition, 2014.

[15] Dmitry Plotnikov, Dmitry Melnik, Mamikon Vardanyan, Ruben Buchatskiy, Roman Zhuykov, and Je-Hyung Lee. Automatic tuning of compiler optimizations and analysis of their impact. *Procedia Computer Science*, 18(0):1312 – 1321, 2013. 2013 International Conference on Computational Science.

[16] William H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes 3rd Edition: The Art of Scientific Computing*. Cambridge University Press, New York, NY, USA, 3 edition, 2007.

[17] Philip C Roth and Barton P Miller. Deep start: a hybrid strategy for automated performance problem searches. In *Euro-Par 2002 Parallel Processing*, pages 86–96. Springer, 2002.

[18] Justin Thiel. An overview of software performance analysis tools and techniques: From gprof to dtrace. *Washington University in St. Louis, Tech. Rep*, 2006.

[19] William Webber, Alistair Moffat, and Justin Zobel. A similarity measure for indefinite rankings. *ACM Trans. Inf. Syst.*, 28(4):20:1–20:38, November 2010.