Final Report
PetaScale Application Development Analysis
Grant Number DE-FG02-04ER25629 A003

Robert W. Numrich *

June 20, 2008

Summary

The results obtained from this project will fundamentally change the way we look at computer performance analysis. These results are made possible by the precise definition of a consistent system of measurement with a set of primary units designed specifically for computer performance analysis. This system of units, along with their associated dimensions, allows us to apply the methods of dimensional analysis, based on the Pi Theorem, to define scaling and self-similarity relationships. These relationships reveal new insights into experimental results that otherwise seems only vaguely correlated. Applying the method to cache-miss data revealed scaling relationships that were not seen by those who originally collected the data. Applying dimensional analysis to the performance of parallel numerical algorithms revealed that computational force is a unifying concept for understanding the interaction between hardware and software. The efficiency of these algorithms depends, in a very intimate way, on the balance between hardware forces and software forces. Analysis of five different algorithms showed that performance analysis can be reduced to a study of the differential geometry of the efficiency surface. Each algorithm defines a set of curvilinear coordinates, specific to that algorithm, and different machines follow different paths along the surface depending on the difference in balance between hardware forces and software forces. Two machines with the same balance in forces follow the same path and are self-similar.

The most important result from the project is the statement of the Principle of Computational Least Action. This principle follows from the identification of a dynamical system underlying computer performance analysis. Instructions in a computer are modeled as a classical system under the influence of computational forces. Each instruction generates kinetic energy during execution, and the sum of the kinetic energy for all instructions produces a kinetic energy spectrum as a function of time. These spectra look very much like the spectra used by chemists to analyze properties of molecules. Large spikes in the spectra reveal events during execution, like cache misses, that limit performance. The area under the kinetic energy spectrum is the computational action generated by the program. This computational action defines a normed metric space that measures the size of a program in terms of its action norm and the distance between programs in terms of the norm of

*rwn/papers/progressReports/petaFinal/petaFinal.tex

the difference of their action. This same idea can be applied to a set of programmers writing code and leads to a computational action metric that measures programmer productivity. In both cases, experimental evidence suggests that highly efficient programs and highly productive programmers generate the least computational action.

1 Scaling

A major goal for this project was the establishment of a consistent set of units and dimensions for measurement of both performance and productivity. A well defined system of measurement is required, first to design reproducible experiments and second to define new metrics to describe the results of those experiments. The results of this project are a significant extension of my previous work on the topic [33, 34, 36].

1.1 A system of measurement

The scientific community has defined the International System of Units (SI) [51, 52] with seven primary units, but the system has not been augmented to include units for computer performance measurements. Prefixes based on powers-of-two, rather than powers-of-ten, have been proposed [31], but defining prefixes different from those used by other disciplines leads to confusion and obfuscates the real issue.

The real issue is the definition of a set of primary units for computer performance analysis. Except for the unit of time, the SI system contains no other primary unit that applies to computer performance analysis. We measure length in bytes not meters. We measure work in floating-point operations not joules. It makes sense, then, to base a system of measurement [36] on the primary units, length in bytes, work in floating-point operations, and time in seconds, as shown in Table (1.1).

In a system of measurement based on these primary quantities, some common quantities that appear in computer performance analysis have the matrix of dimensions,

	L (byte)	E (flop)	T (s)
length	1	0	0
work (energy)	0	1	0
time	0	0	1
frequency	0	0	-1
clock period	0	0	1
velocity (bandwidth)	1	0	-1
force (intensity)	-1	1	0
power	0	1	-1
mass	-2	1	2
momentum	-1	1	1
action	0	1	1

. (1.1)

Computational power, a derived quantity, is measured in units of floating-point operations per second, and bandwidth is measured in units of velocity, bytes per second. The quantity commonly called computational intensity [24, 30, 36, 43, 40] is measured in units of force, floating-point operations per byte. Computational mass, momentum and action are also well defined quantities as shown

in the table. At times it is useful to use the clock period as the unit of time where 1 cp $= \nu^{-1}$ s is the reciprocal of the machine frequency.

It is certainly possible to pick other primary units for a system of measurement. The only requirement is that they be used correctly within the system of measurement they define.

1.2 Dimensional analysis

Many of the results obtained during this project depend on dimensional analysis and the application of its fundamental tool, the Pi Theorem. The Pi Theorem states that, for each particular problem, there exists a set of scale factors such that all variables become dimensionless and the number of independent variables is less than the original number. This delicate and beautiful result is based on the simple fact that the universe doesn't care what system of measurement is used to observe it. In other words, the functional relationships among physical quantities are covariant with respect to changes in units. References [9, 10, 11, 16, 17] discuss dimensional analysis in general and the Pi Theorem in particular. References [5, 6, 7, 26] discuss dimensional analysis in the special setting of electromagnetism.

The following description of the subject paraphrases the algebraic approach of Birkhoff [8, Ch. IV] and others [9, 16]. To establish a system of measurement, first pick k positive primary units,

$$u = (u_1, u_2, \ldots, u_k) \ , \qquad u_i > 0 \ , \quad i = 1, \ldots, k \ . \tag{1.2}$$

In mechanics, for example, the primary units are length, with dimension L and value $u_1 = 1$ m; mass, with dimension M and value $u_2 = 1$ kg; and time, with dimension T and value $u_3 = 1$ s. For computer performance analysis, a good choice for the three primary units are length, with dimension L and value $u_1 = 1$ byte; work, with dimension E and value $u_2 = 1$ flop; and time, with dimension T and value $u_3 = 1$ s.

Given a set of n physical quantities

$$X = (x_1, x_2, \ldots, x_n) \ , \tag{1.3}$$

the matrix of dimensions, in this system of measurement, is the $n \times k$ matrix,

$$\begin{array}{c|cccc} & u_1 & u_2 & \cdots & u_k \\ \hline x_1 & d_1^1 & d_1^2 & \cdots & d_1^k \\ x_2 & d_2^1 & d_2^2 & \cdots & d_2^k \\ x_3 & d_3^1 & d_3^2 & \cdots & d_3^k \\ \vdots & \vdots & \vdots & \cdots & \vdots \\ x_n & d_n^1 & d_n^2 & \cdots & d_n^k \end{array} \tag{1.4}$$

with element d_j^i equal to the dimension of quantity x_j with respect to unit u_i. Dimensional analysis is based on the algebraic properties of this matrix of dimensions. Each physical quantity x_j has a numerical value $\hat{x}_j$ relative to this system of units such that

$$x_j = \hat{x}_j \ u_1^{d_j^1} u_2^{d_j^2} \cdots u_k^{d_j^k} \tag{1.5}$$

with each unit raised to a power equal to its dimension d_j^i.

Dimensional analysis is the study of scaling. Given the k positive scale factors,

$$\alpha = (\alpha_1, \alpha_2, \ldots, \alpha_k) \ , \qquad \alpha_i > 0 \ , \quad i = 1, \ldots, k \ , \tag{1.6}$$

and the matrix of dimensions (1.3), define the linear operator [8, Ch. IV],

$$S_\alpha x_j = \alpha_1^{d_j^1} \alpha_2^{d_j^2} \cdots \alpha_k^{d_j^k} x_j \ . \tag{1.7}$$

or using the compact notation in terms of point-wise exponentiation,

$$S_\alpha x_j = \alpha^{d_j} x_j \ . \tag{1.8}$$

As noted by Birkhoff [8], these operators represent the multiplicative group on the set of positive vectors α of length k.

The Pi Theorem applies to unit-free relationships according to the following definition.

Definition 1 (Unit-free relationships (Birkhoff [8, p.89])) *A functional relationship between quantities,*

$$F(x_1, x_2, \ldots, x_n) = 0 \tag{1.9}$$

is **unit-free** *if (1.9) implies*

$$F(S_\alpha x_1, S_\alpha x_2, \ldots, S_\alpha x_n) = 0 \tag{1.10}$$

for all choices of the scaling parameters $\alpha > 0$.

Since relationship (1.10) holds for all choices of the scaling parameters, pick them in such a way that

$$S_\alpha x_j = 1 \ , \qquad j = 1, k \ , \tag{1.11}$$

for k dimensionally independent quantities, permuted to the first k rows in the dimension matrix if necessary. Take the logarithm of both sides of (1.11), using definition (1.7) for the scaling operators, to obtain the system of equations,

$$\begin{bmatrix} d_1^1 & d_2^1 & \cdots & d_k^1 \\ d_1^2 & d_2^2 & \cdots & d_k^2 \\ \vdots & \vdots & \cdots & \vdots \\ d_1^k & d_2^k & \cdots & d_k^k \end{bmatrix} \begin{bmatrix} \log(\alpha_1) \\ \log(\alpha_2) \\ \vdots \\ \log(\alpha_k) \end{bmatrix} = \begin{bmatrix} \log(1/x_1) \\ \log(1/x_2) \\ \vdots \\ \log(1/x_k) \end{bmatrix} \ . \tag{1.12}$$

As long as the quantities x_i are dimensionally independent, that is, as long as the matrix on the left side of (1.12) has rank k, there is a unique solution for the scaling parameters α_i. The reciprocals of the scale factors define a new set of primary units for a new system of measurement.

Using these scale factors, define the dimensionless self-similarity parameters,

$$\pi_j = S_\alpha x_{k+j} \ , \qquad j = 1, n-k \ . \tag{1.13}$$

With these definitions, the relationship (1.10) assumes the form

$$\begin{aligned} &F(S_\alpha x_1, S_\alpha x_2, \ldots, S_\alpha x_k, S_\alpha x_{k+1}, \ldots, S_\alpha x_n) \\ &\qquad = F(1, 1, \ldots, 1, \pi_1, \pi_2, \ldots, \pi_{n-k}) = 0 \ . \end{aligned} \tag{1.14}$$

Define the function,

$$\Phi(\pi_1, \pi_2, \ldots, \pi_{n-k}) = F(1, 1, \ldots, 1, \pi_1, \pi_2, \ldots, \pi_{n-k}) \ , \tag{1.15}$$

and state the Pi Theorem.

Theorem 1.1 (Pi Theorem ([8, p. 93])) *Let n positive quantities $X = (x_1, \ldots, x_n)$ transform under all changes $\alpha > 0$ in the k primary units $u = (u_1, \ldots, u_k)$ according to (1.7). Let the $n \times k$ matrix of dimensions (1.4) have rank k. Then any unit-free relationship of the form*

$$F(x_1, \ldots, x_n) = 0 \tag{1.16}$$

is equivalent to a relationship of the form

$$\Phi(\pi_1, \ldots, \pi_{n-k}) = 0 \tag{1.17}$$

for suitable self-similarity parameters of definition (1.13).

2 Cache behavior

Analysis of the cache-miss ratio as a function of cache size, line size and degree of associativity provides a good example of how to apply dimensional analysis. The cache-miss ratios measured by Agarwal, Horowitz and Hennessy [1] provide a specific example. In their paper, they used two different units of length, bytes and words, where 1 word = 4 byte, requiring extreme care to interpret their results. They presented cache-miss ratios averaged over eight programs with average trace length, $n = 356000$ word and average program working set size, $m = 7816$ word. In their Figure 5, they presented cache-miss ratios for two line sizes, $l = 4$ byte and $l = 16$ byte, both for a direct-mapped cache, with $d = 1$, and for a two-way set-associative cache, with $d = 2$, for several cache sizes from $c = 2^{10}$ byte to $c = 2^{18}$ byte. The number of cache misses can be extracted, with some effort, from their figure by reading the logarithm of the miss ratio, exponentiating, and multiplying by the average trace length. The uncertainty in the raw data extracted this way is large because the uncertainty in their original data is compounded with the uncertainty of reading it from their figure.

If the number of cache misses,

$$\chi = \chi(m, c, l, d) \ , \tag{2.1}$$

defines a unit-free relationship as a function of the four variables m, c, l and d, with the matrix of dimensions,

	χ	n	m	c	l	s	d
L	1	1	1	1	1	1	0

, (2.2)

the choice of scaling parameter for length such that $\alpha_L l = 1$ implies the scaled relationship,

$$\chi/l = \chi(m/l, k, 1, d) \ . \tag{2.3}$$

The ratio,

$$k = c/l \ , \tag{2.4}$$

is the cache shape or the number of lines in the cache.

It is reasonable to expect the number of misses to increase with increasing values of the program footprint m and to decrease with increasing values of the line size l and with increasing values of the set-associativity parameter d. Since the number of misses often exhibits an inverse power law with respect to cache size, the heuristic,

$$\chi/l = d^{-\gamma} \beta^2 (m/l)^2 k^{-\alpha} \ , \tag{2.5}$$

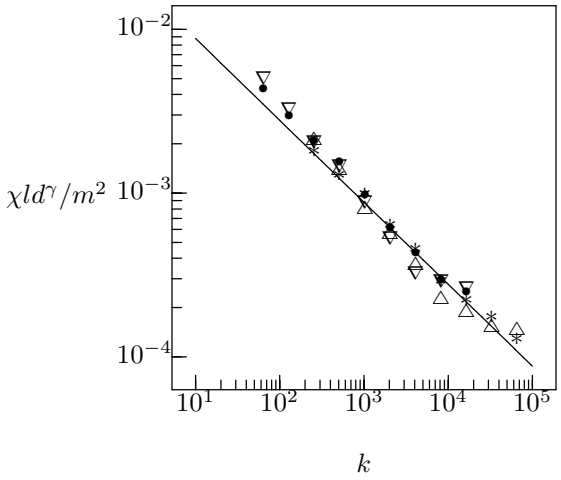

Figure 1: Cache-miss data from [1, Figure 5] scaled according to formula (2.6). The descending solid line is the straight line predicted for the log-log plot (2.7) with $\alpha = 1/2$, $\beta = 1/6$ and $\gamma = 1/3$. The bullets ($\bullet$) represent Agarwal's data for $l = 16$ byte and the crosses ($\oplus$) represent data for $l = 4$ byte for a direct-mapped cache, $d = 1$. The grads (∇) represent Agarwal's data for $l = 16$ byte and the triangles ($\triangle$) represent data for $l = 4$ byte for a two-way set-associative cache, $d = 2$.

with three constants α, β and γ independent of k and m/l, is a reasonable place to start. If this heuristic works, the scaled number of misses,

$$\chi l d^\gamma/m^2 = \beta^2 k^{-\alpha} \ , \tag{2.6}$$

should yield a straight line on a log-log plot,

$$\log(\chi l d^\gamma/m^2) = -\alpha \log(k) + \log(\beta^2) \ , \tag{2.7}$$

with slope $-\alpha$ and intercept $\log(\beta^2)$. Indeed, the log-log plot of Agarwal's scaled data, shown in Figure 1, yields a straight line with the parameters $\alpha = 1/2$, $\beta = 1/6$ and $\gamma = 1/3$.

Let $\chi(k)$ be the number of cache misses at cache shape k. The function $\chi(k)/l$ of formula (2.5) decreases with k and has a fixed point,

$$\chi(k_0)/l = k_0 \ , \tag{2.8}$$

where the number of line misses equals the cache shape. From (2.5), the cache shape k_0 is determined by the formula,

$$k_0^{1+\alpha} = d^{-\gamma}\beta^2 (m/l)^2 \ . \tag{2.9}$$

The number of cache misses, then, at cache shape k has the simple form,

$$\chi(k) = \chi(k_0) \left(\frac{k}{k_0}\right)^{-\alpha} \ , \tag{2.10}$$

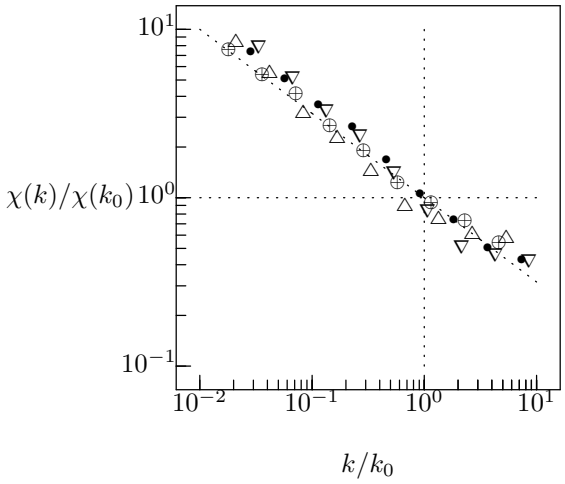

Figure 2: Cache-miss data from [1, Figure 5]. The descending dotted line is formula (2.10) with $\alpha = 1/2$, and k_0 is the fixed point cache shape from (2.9). The bullets ($\bullet$) represent Agarwal's data for $l = 16$ byte and the crosses ($\oplus$) represent data for $l = 4$ byte for a direct-mapped cache, $d = 1$. The grads ($\triangledown$) represent Agarwal's data for $l = 16$ byte and the triangles ($\triangle$) represent data for $l = 4$ byte for a two-way set-associative cache, $d = 2$.

relative to the number of cache misses at cache shape k_0.

Figure 2 shows that the heuristic (2.5) works very well. All of Agarwal's data [1, Figure 5] for two different line sizes, $l = 4$ byte and $l = 16$ byte, and for two degrees of set associativity, $d = 1$ and $d = 2$, fall along the same line with slope equal to -1/2. The scatter in the data is of the same order of magnitude as the scatter in Agarwal's original data, and considering the difficulty of extracting data from the original logarithmic plots, the agreement of the data with the scaled model is quite remarkable.

The cache shape $k_0 = c_0/l$ at the fixed point provides a criterion for the critical cache size for a given line size. Indeed, substitution of the values, $\alpha = 1/2$, $\beta = 1/6$ and $\gamma = 1/3$ in formula (2.9) yields

$$c_0 = d^{-2/9} l^{-1/3} (m/6)^{4/3} \ . \tag{2.11}$$

Notice that the combination of exponents, $4/3 - 1/3 = 1$, gives this formula the correct dimension of length. For smaller caches, the number of line misses is greater than the cache shape. For larger caches, the number is smaller than the cache shape. For line size $l = 32$ byte and average working set size $m = 4 \times 7816$ byte, the critical cache size has the value, $c_0 \approx 28.5 \times 10^3$ byte for a direct-mapped cache, $d = 1$, and $c_0 \approx 24.4 \times 10^3$ byte for a degree-two cache, $d = 2$.

Finally, formula (2.5) yields the cache-miss ratio,

$$\chi/n = d^{-\gamma} \beta^2 \left(\frac{m^2}{l \cdot n} \right) k^{-\alpha} \ . \tag{2.12}$$

Since $\alpha = 1/2$, the dependence of this formula on the square root of the cache shape agrees with the speculation of Hartstein and coworkers [22], although the next example shows that this dependence is by no means the same for all cases. The formula predicts that the cache-miss ratio decreases to zero for long traces if the working set grows more slowly than the square root of the trace length, $m(n) < \sqrt{ln}$, for large n.

As a second example, consider a model for the number of cache misses presented by Singh, Stone and Thiébaut [49]. They extended a previous model that considered the behavior as a function of cache size with fixed line size $l = 1$ byte due to Thiébaut [53, 54]. Their new model includes the effect of both cache size and line size for fully-associative caches such that $d = c/l$.

Without going into as much detail for their model, it is possible to show that the number of cache misses obeys the relationship,

$$\chi(k) = \chi(k_0) \left(\frac{k}{k_0} \right)^{1-\theta} , \tag{2.13}$$

where

$$k_0^{1-\theta} = (A/l^{\alpha})^{-\theta} . \tag{2.14}$$

The parameter A is related to the footprint of the program and the parameter θ is called the fractal dimension of the program.

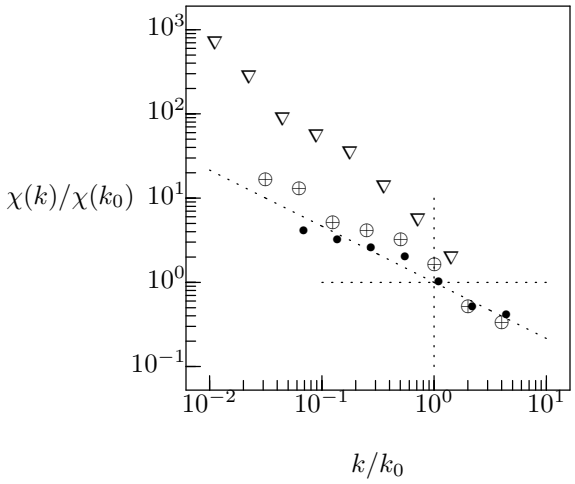

Figure 3: Scaled cache-miss data from [49, Figure 9]. The three sets of data represent different line sizes: ($\bullet$) for $l = 16$ byte; ($\oplus$) for $l = 128$ byte; and (∇) for $l = 2048$ byte. The value $\chi(k_0)$ equals the number of misses at the critical cache shape $k = k_0$ The descending dotted line is the asymptotic cache-miss function of (2.13) with slope $1 - \theta = -2/3$. As described in the text, the measurements for the case $l = 2048$ byte lie outside the range of validity for the model and are not expected to lie on the dotted line.

Figure 3 shows cache-miss data extracted from Singh [49, Figure 9]. On a log-log plot, the scaled miss data as a function of the scaled cache shape should be a straight line with slope -2/3. Singh comments on the fact that the model reproduces the measured data only for large enough caches. My analysis shows that the model should not be expected to work for cache shapes $k < k_0$. In fact, the critical shape for $l = 2048$ is outside the region where the model applies. Most of the measured data represents points below the critical cache shape. But formula (2.13) clearly represents the asymptotic behavior of the number of misses for cache shapes greater than the critical cache shape. All three sets of measured data look as if they approach the asymptote for large values of k/k_0, but more measurements are needed to decide the issue. An important advantage provided by dimensional analysis is that it provides absolute criteria for what measurements to make to validate a particular model.

A major goal of dimensional analysis is to identify and state self-similarity relationships. The two case studies for cache behavior can be summarized as a self-similarity relationship that shows how the two cases are similar while the specific details for each case are different.

Self-Similarity Relationship 2.1 *Let the number of cache misses, $\chi(k)$, for cache shape $k = c/l$, be defined by (2.5) for the Agarwal data and by (2.13) for the Singh data. Let the critical cache shape k_0 by defined by (2.9) for Agarwal and by (2.14) for Singh. Furthermore, let the exponent $\gamma = 1/2$ for Agarwal and $\gamma = 2/3$ for Singh. Then the commutator,*

$$[\chi(k), k_0] = \chi(k)k_0^{-\gamma} - k^{-\gamma}\chi(k_0) \ , \tag{2.15}$$

is zero,

$$[\chi(k), k_0] = 0 \ . \tag{2.16}$$

3 Computational forces in parallel algorithms

Another important result that follows from dimensional analysis is the identification of computational force as a unifying idea for the analysis of performance for parallel numerical algorithms. I have applied the analysis to several different algorithms starting with very simple algorithms and progressing to more complicated, full applications.

3.1 Parallel matrix multiplication

The execution time for parallel matrix multiplication [18, 38] is the sum of three terms,

$$t(m_1, m_2, b_1, b_2, w, r_0) = w/r_0 + m_1/b_1 + m_2/b_2 \ . \tag{3.1}$$

For matrices of size $n \times n$ on a machine with $p \times p$ processors with a blocked data distribution, each processor holds a block of size $n/p \times n/p$ and performs

$$w = 2n(n/p)^2 e_0 \tag{3.2}$$

floating-point operations where e_0 is the unit of work. For the algorithm described in a previous paper [38, equation (5.2.5)], each processor moves three blocks from local memory,

$$m_1 = 3(n/p)^2 l_0 \ , \tag{3.3}$$

where l_0 is the unit of length, and $2(p-1)$ blocks from remote memory,

$$m_2 = 2(p-1)(n/p)^2 l_0 \ , \tag{3.4}$$

one from each of the $p-1$ processors in the same row and another one from each of the $p-1$ processors in the same column. If there is no overlap between these three phases of the computation, the execution time is the sum of three terms,

$$t = \frac{2n(n/p)^2}{r_0} e_0 + \frac{3(n/p)^2}{b_1} l_0 + \frac{2(p-1)(n/p)^2}{b_2} l_0 \ , \tag{3.5}$$

obtained by substituting (3.2)-(3.4) into (3.1).

Define the locality and bandwidth parameters,

$$\lambda = 3/(2(p-1)) \ , \qquad \beta = b_1/b_2 \ , \tag{3.6}$$

and the granularity parameters,

$$\Gamma = \begin{bmatrix} (\phi_3/f_0)/(2n/3) & (\phi_3/f_0)/(n/(p-1)) \\ (\phi_4/f_0)/(2n/3) & (\phi_4/f_0)/(n/(p-1)) \end{bmatrix} \ , \tag{3.7}$$

where $\phi_3 = r_0/b_1$ and $\phi_4 = r_0/b_2$, are the hardware forces involved, and $f_0 = e_0/l_0$ is the unit of force. Then the surface

$$e_\lambda = \frac{1}{1 + \lambda\gamma_{12} + \gamma_{22}} \ , \tag{3.8}$$

is the efficiency surface parameterized by the locality parameter $\lambda(p)$, a property of the algorithm alone independent of any particular machine. It depends on the number of processors, but is independent of the problem size. All machines with the same number of processors lie on the same surface. For the same problem size, however, each machine lies at a different point on the surface depending on the specific values of its hardware forces. Clearly, since the software forces in the denominators of the granularity parameters grow linearly with the problem size, the programmer can overcome the hardware forces on any machine by picking a large enough problem.

Performance analysis reduces to a study of the differential geometry of this self-similarity surface [55, Ch. II]. Define two curvilinear coordinates, $u = \gamma_{12}$ and $v = \gamma_{22}$, with values from (3.7),

$$u(n) = (\phi_3/f_0)/(n(p-1)) \tag{3.9}$$

$$v(n) = (\phi_4/f_0)/(n(p-1)) \ , \tag{3.10}$$

functions of the problem size n for fixed number of processors p. As the problem size changes, each machine follows a path along the surface, described by the vector,

$$\mathrm{r}(n) = [u(n), v(n), e_\lambda(u(n), v(n))] \ , \tag{3.11}$$

with

$$e_\lambda(u(n), v(n)) = \frac{1}{1 + \lambda u(n) + v(n)} \ . \tag{3.12}$$

The two coordinates are related by the bandwidth ratio,

$$v = \beta u \ , \tag{3.13}$$

as can be seen by dividing (3.10) by (3.9) and comparing with definition (3.6). The efficiency curve, then, can be written as a function of n and β,

$$e_\lambda(u(n), \beta(u(n)) = \frac{1}{1 + \lambda u(n) + \beta u(n)} \ , \tag{3.14}$$

for fixed $\lambda(p)$.

As shown in Figure 4, each machine with the same value of β follows the same path on the surface as the problem size n changes. Since $u(n)$ by definition (3.9) decreases as n increases, the path approaches unity for very large problems. Larger values of β push the curves down the surface to lower efficiencies. For fixed problem size, paths for different values of β cut across these first paths from high efficiency for low β to low efficiency for high β. Low values of β correspond to high bandwidth from secondary memory relative to bandwidth from primary memory.

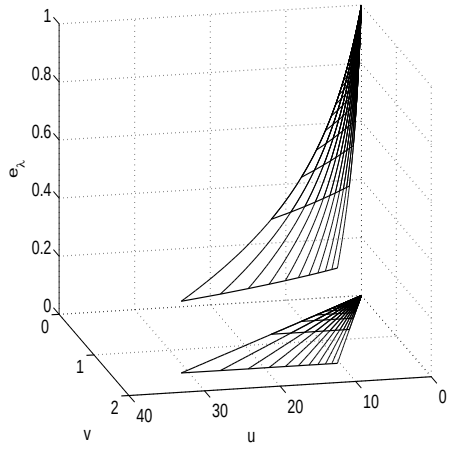

Figure 4: The self-similarity surface for parallel matrix multiplication using $\lambda(p)$ as the surface parameter for fixed $p = 32$ or $p^2 = 1024$ processors. Any machine with the same bandwidth ratio β approaches the summit along the same path as the problem size increases. Higher values of β define paths lower on the surface. The projection of these paths onto the (u, v) plane are straight lines with slope β. The machine with smaller β, that is, higher remote bandwidth relative to local bandwidth, has an easier ascent to the summit. For fixed problem size, paths cut across the first set of paths, higher to lower, as the value of β increases.

For fixed problem size, the efficiency hops from one surface to another as the number of processors changes. Figure 5 shows the decrease in efficiency for a fixed problem size $n = 1000$ for $p = 1, 4, 8, 16, 32, 64, 128$. The bullets mark the points on the surfaces, but the surfaces themselves are not shown for clarity of presentation. The projection of these points onto the (u, v) plane is a straight line with slope β as can be seen by eliminating p from (3.9) for u and (3.10) for v at fixed n.

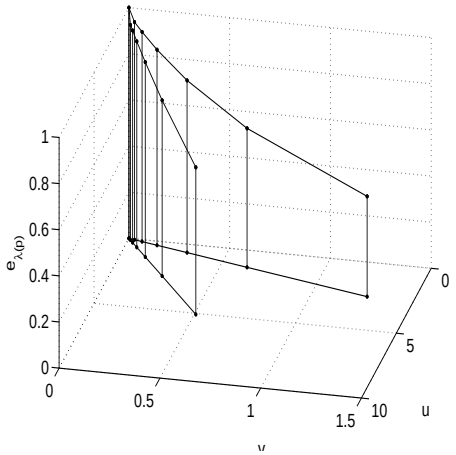

Figure 5: Efficiency for matrix multiplication for fixed problem size $n = 1000$ as the the number of processors increases, $p = 1, 4, 8, 16, 32, 64, 128$. The efficiency jumps from one surface to another, parameterized by $\lambda(p)$, as the number of processors changes. The surfaces are not shown for clarity of presentation. The projection onto the (u, v) plane is a straight line with slope $\beta = b_1/b_2$. Two machines are shown, one with $\beta = 0.1$ and one with $\beta = 0.5$. Higher bandwidth b_2 from remote memory relative to bandwidth b_1 to local memory results in lower values for β and consequently higher efficiency as the number of processors p increases.

3.2 Other parallel algorithms

I have applied dimensional analysis to several other algorithms to show its generality. I started with very simple programs with simple timing formulas to gain confidence in the methodology [40]. I then considered increasingly more complicated timing formulas [41, 42, 43] to show that the methodolgy does not depend on the simplicity of the formula nor on its being continuous or differentiable.

Figure 6, for example, shows the results for a generic algorithm based on the work of Stewart [50]. If we use the same numerical algorithm, the software force ϕ_0 has not changed in the decade and a half since the publication of Stewart's paper. But the hardware forces ϕ_1 and ϕ_2 have changed as shown in the following table,

	Machine 1 ca. 1990	Machine 2 ca. 2007
$1/\alpha$	10^6 flop/s	10^9 flop/s
σ	10^{-4} s	10^{-6} s
$1/\tau$	10^6 word/s	10^9 word/s
ϕ_1	10^2 flop/word	10^3 flop/word
ϕ_2	1 flop/word	1 flop/word

(3.15)

Because of these changes, modern machines follow paths lower on the efficiency surface than paths followed by older machines.

Figure 7 shows a similar result for the parallel QR algorithm analyzed by Henry and van de

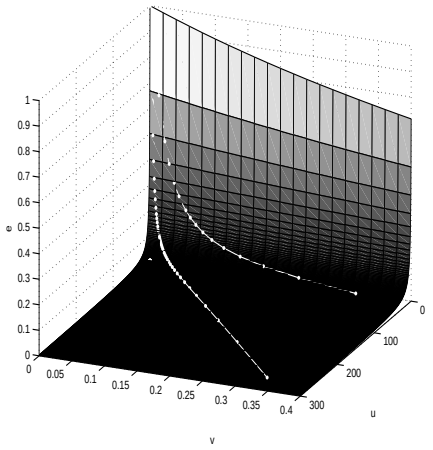

Figure 6: Two paths along the efficiency surface for Stewart's generic algorithm for fixed number of processors, $p = q^2 = 32^2 = 1024$. The higher path corresponds to the older Machine 1; the lower path to the modern Machine 2. The problem size increases from $n = 500$ at low efficiency in increments of 100. In the limit of very large problem size, both machines approach perfect efficiency. They approach the limit at different rates along different paths on the surface.

Geijn [23]. The following table, appropriate for their algorithm,

	Machine 1 ca. 1990	Machine 2 ca. 2007
α	10^{-4} s	10^{-6} s
β^{-1}	10^6 word/s	10^9 word/s
γ^{-1}	10^6 flop/s	10^9 flop/s
ϕ_1	10^2 flop/word	10^3 flop/word
ϕ_2	1 flop/word	1 flop/word

, (3.16)

shows the same change in hardware forces ϕ_1 and ϕ_2 over approximately the last two decades. Figure 11 shows two paths along the efficiency surface, one for each machine in the table. The number of processors is fixed at $p = 512$, and each point on a path corresponds to a different problem size calculated from the curvilinear coordinates $u_*(n)$ and $v_*(n)$ from equation (6.25). For large enough problem size, both machines approach perfect efficiency. But they follow different paths at different rates along the surface to reach the summit. The modern machine is clearly less efficient than the earlier machine.

A third example is the Linpack benchmark analyzed by Greer and Henry [21]. In the ten years since they described the ASCI Red machine, hardware forces have changed. For example, Table 3.17 compares the ASCI Red machine to the Cray XT3, with values for the hardware parameters G, B and S reported for the Cray machine on the HPC Challenge website [25], and to the IBM Blue

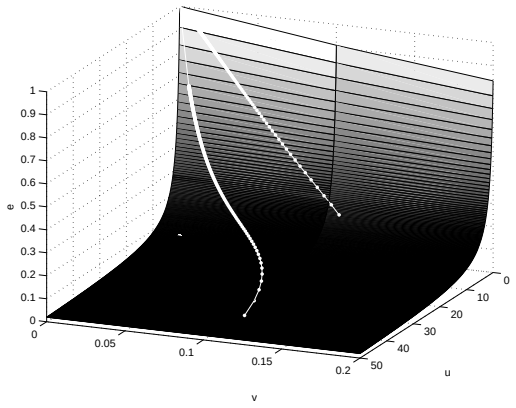

Figure 7: Two paths along the efficiency surface for the QR algorithm for fixed number of processors, $p = 512$. The problem size increases from $n = 500$ at low efficiency in increments of 100. In the limit of very large problem size, both machines approach perfect efficiency. They approach the limit at different rates along different paths on the surface. The higher path on the surface corresponds to Machine 1; the lower path to Machine 2.

Gene/L, with values from the same website.

	ASCI Red (1997) [21]	Cray XT3 (2007) [25]	IBM Blue Gene/L (2007) [25]
G	3.3×10^8 flop/s	44×10^8 flop/s	19×10^8 flop/s
B	3.9×10^8 byte/s	11×10^8 byte/s	1.6×10^8 byte/s
S	30×10^{-6} s	21×10^{-6} s	7.1×10^{-6} s
ϕ_1^H	0.85 flop/byte	4.0 flop/byte	12 flop/byte
ϕ_2^H	9.9×10^3 flop/byte	92×10^3 flop/byte	13×10^3 flop/byte
p	16	65	64
q	286	80	1024
k	64	60	192
$l_{1/2}$	1.2×10^4 byte	2.3×10^4 byte	0.11×10^4 byte
$\hat{n}$	3.0×10^4 byte	3.7×10^4 byte	1.0×10^4 byte
$\hat{e}$	0.82	0.35	0.25

(3.17)

The software forces have not changed because the algorithm today is much the same as ten years ago. In fact, their formula, even using the ASCI Red values for α, β and γ, predicts an execution time within 3% of the measured value for the Cray machine. The formula is only within 31% of the measured value for the IBM machine implying either that the three parameters α, β and γ are different for that machine or that a different algorithm has been used.

The hardware forces, on the other hand have changed. The modern machines follow paths along the efficiency surface quite different from the path followed by the older machine as shown in Figure 8. For large problem sizes, all machines approach perfect efficiency. But the modern machines have a harder ascent to the summit than the older machine.

The final example is the SAGE benchmark analyzed by Kerbyson and coworkers [27]. Figure

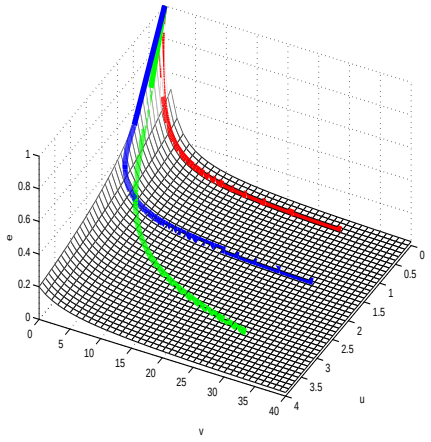

Figure 8: Efficiency as a function of problem size for the Linpack benchmark. The topmost path, marked in red, corresponds to the ASCI Red machine from the first column of Table 3.16 decomposed as $(p, q) = (16, 286)$ with block size $k = 64$; the bottommost path, marked in green, to the Cray XT3 from the second column of Table 3.16 decomposed as (65,80) with $k = 60$; and the middle path, marked in blue, to the IBM Blue Gene/L from the third column decomposed as (64,1024) with $k = 192$. All three machines reach high efficiency for large problem sizes. But they approach the summit along quit different paths determined by the differences in their hardware forces.

9 shows the paths for the four machines studied in the original paper [27]. The red line down the center of the surface divides it into two halves. A path down the surface on the left side of the red line corresponds to a machine limited by latency. A path down the surface on the right side of the red line corresponds to a machine limited by bandwidth.

The four machines separate into three groups. The two Alpha Servers ES40 and ES45 are self-similar. They follow the same path on the surface marked by the yellow line (ES45) and the green line (ES40), which fall on top of each other. These two machines are bandwidth limited. The ASCI Blue machine follows the blue line. It is latency limited for small values of p but becomes bandwidth limited for large values of p. The ASCI White machine follows the white line. It is bandwidth limited much like the Alpha Server machines, but it is also latency limited for small values of p, although less so than the ASCI Blue machine.

3.3 New performance metrics

Since the performance analysis for each of these algorithms reduces to paths followed along an efficiency surface, the differential geometry of this surface suggests several new performance metrics. For example, for matrix multiplication, each machine approaches the peak at its own angle defined by its tangent,

$$\tan\theta = \beta \ . \tag{3.18}$$

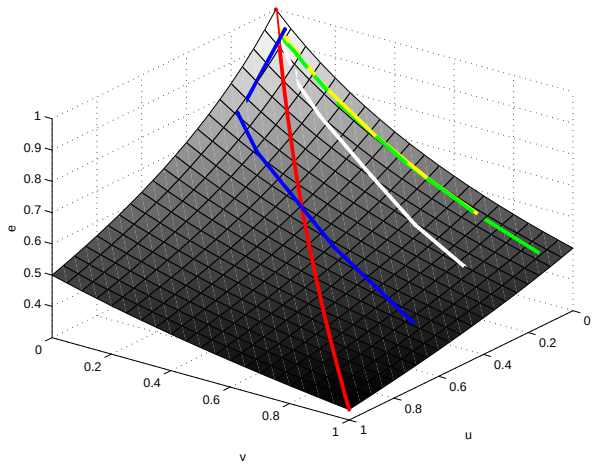

Figure 9: Paths on the efficiency surface for the SAGE benchmark as a function of the number of processors. The red line divides the surface into a latency-limited half on the left as the coordinate u increases and a bandwidth-limited half on the right as the coordinate v increases. The Alpha Server ES45 follows the yellow path; the Alpha Server ES40 follows the green path, which falls on top of the yellow path; the ASCI White machine follows the white path; and the ASCI Blue machine follows the blue path.

A measure of the difference between two machines is the difference between their angles of approach,

$$\tan(\theta_i - \theta_j) = \frac{\tan\theta_i - \tan\theta_j}{1 + \tan\theta_i \tan\theta_j} = \frac{\beta_i - \beta_j}{1 + \beta_i\beta_j} \ . \tag{3.19}$$

Two machines with the same value of β are equivalent and approach the summit at the same angle.

Another way to measure the difference between machines is to examine the tangent vector dr/dn along the path to the summit. The distance from the smallest problem $n = p$ to the hypothetical infinite problem at the peak is the integral along the surface,

$$s(p) = \int_p^\infty \sqrt{|dr/dn|^2}\, dn \ , \tag{3.20}$$

of the magnitude of the tangent vector [55, Ch. II].

Alternative measures include the distance along the geodesic between points with equal values of n or the area on the surface between two curves or the area between the triangles projected to the (u, v) plane.

4 Fixed-time benchmarks

Another application of dimensional analysis resulted in a new way to look at the Linpack benchmark [39]. Without going into details, Figure 10 shows that it is possible to pick a fixed time,

to determine a problem size for each processor decomposition (p, q) that runs in that fixed time, and to determine two dimensionless self-similarity parameters (α, β) that are characteristic of each particular machine. As Figure 10 shows, the analysis made the assymmetry of the algorithm with respect to processor decomposition very obvious. This assymmetry is not obviously apparent from the usual way of analyzing this benchmark. I also applied the fixed-time constraint to the SAGE benchmark [42].

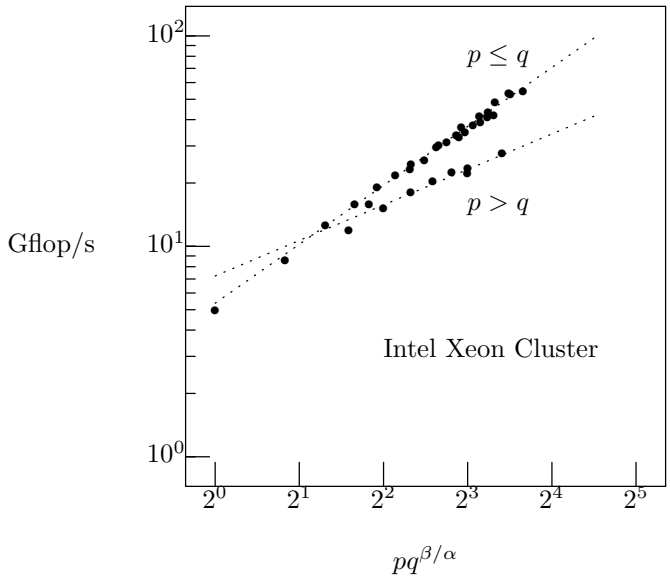

Figure 10: Computational power as a function of the number of processors for a Intel Xeon cluster. The bullets represent measured values. The dotted lines are the least-squares fits of the measured values.

5 Computational energy spectra

In a series of papers [35, 44, 46], I have described computer performance analysis in terms of the computational energy spectrum of a program as it executes. This spectrum shows what happens as a program executes, clock-tick-by-clock-tick, instruction-by-instruction. Characteristic spikes in the spectrum correspond to important events, such as cache misses, that limit performance. Measuring the spectrum requires observation of just two pieces of information, the clock-tick when each instruction issues and the clock-tick when each instruction completes. This information can be obtained as the program executes in real time or from an execution trace processed through an instruction-level simulator.

The area under the energy spectrum is the computational action generated during execution.

The action defines a norm that measures a program's size, and this norm defines a metric space with a distance function between programs. The energy spectra measured so far support the conjecture that the program that generates the least action is the best program. No proof of the Principle of Computational Least Action yet exists, but the evidence so far supports the conjecture.

5.1 A dynamical system

Instruction execution can be described as a dynamical system [35, 36, 44, 46] determined by a hamiltonian function,

$$H(r,p) = \frac{p^2}{2m} + V(r) \ , \tag{5.1}$$

and the related equations of motion,

$$\frac{dr}{dt} = \frac{\partial H}{\partial p} = \frac{p}{m} \ , \tag{5.2}$$

$$\frac{dp}{dt} = -\frac{\partial H}{\partial r} = -\frac{dV}{dr} \ , \tag{5.3}$$

with appropriate boundary conditions. The function $r(t)$ is the position of a particle in an abstract computational space as a function of time. The function $p(t)$ is its conjugate momentum.

The total energy is set to zero. The initial boundary condition,

$$r(0) = 0 \ , \qquad p(0) = 0 \ , \tag{5.4}$$

corresponds to a particle at rest at the origin at time $t = 0$, and the final boundary condition,

$$r(\tau) = l \ , \qquad p(\tau) = 0 \ , \tag{5.5}$$

corresponds to a particle again at rest at final position $r = l$ at time $t = \tau$. Under these constraints, the motion is determined by the differential equation,

$$\frac{dr}{dt} = \sqrt{-2V(r)/m} \ , \tag{5.6}$$

with the potential,

$$V(r) = 4e_0[(r/l)^2 - r/l] \ . \tag{5.7}$$

The solution to this equation is the trajectory,

$$r(t) = (l/2)[1 - \cos(\nu t)] \ , \tag{5.8}$$

and the boundary conditions determine the mass,

$$m = \frac{8e_0\tau^2}{\pi^2 l^2} \ . \tag{5.9}$$

The momentum is the mass times the velocity,

$$p(t) = (4e_0\tau/l\pi)\sin(\nu t) \ , \tag{5.10}$$

and the force,

$$f(r) = -8(r/l - 1/2) \cdot (e_0/l) \ , \tag{5.11}$$

is the negative derivative of the potential.

It is convenient to measure time in dimensionless clock-ticks. If the machine has frequency ν_0, time t corresponds to clock-tick,

$$k = \nu_0 t \ , \tag{5.12}$$

and an instruction that completes in τ seconds completes in

$$\kappa = \nu_0 \tau \tag{5.13}$$

clock-ticks.

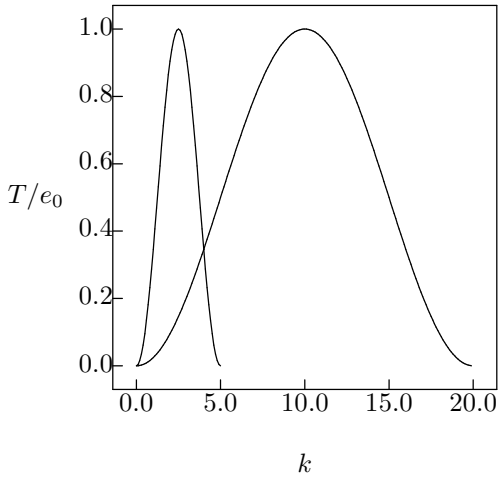

Figure 11: Kinetic energy as a function of time. The curve on the left corresponds to a load from near memory, such as a cache, that takes $\kappa = 5$ clock-ticks. The curve on the right corresponds to a load from far memory that takes $\kappa = 20$ clock-ticks.

The kinetic energy assumes the form,

$$T/e_0 = \sin^2(\pi k/\kappa) \ . \tag{5.14}$$

As a function of time, it provides a picture of how an instruction executes. Figure 11, for example, shows the kinetic energy for two different load instructions. The memory address for one of them hits in cache, and the instruction completes in $\kappa = 5$ clock-ticks. The address for the other one misses cache, and the instruction completes in $\kappa = 20$ clock-ticks. Halfway through execution, the kinetic energy equals the positive unit of energy e_0 matching the negative of the potential energy (5.7) at that point.

Twice the area under the kinetic energy curve,

$$S(k) = 2e_0 \int_0^k \sin^2(\pi z/\kappa) dz \ , \tag{5.15}$$

equals the computational action [3, 20, 29, 44, 35, 37] generated by an instruction at clock-tick k. Evaluation of the integral yields the value,

$$\nu_0 S(k)/e_0 = (\kappa/\pi) \left[\pi k/\kappa - \sin(\pi k/\kappa) \cos(\pi k/\kappa) \right] \ , \tag{5.16}$$

with the total action generated at completion,

$$\nu_0 S(\kappa)/e_0 = \kappa \ . \tag{5.17}$$

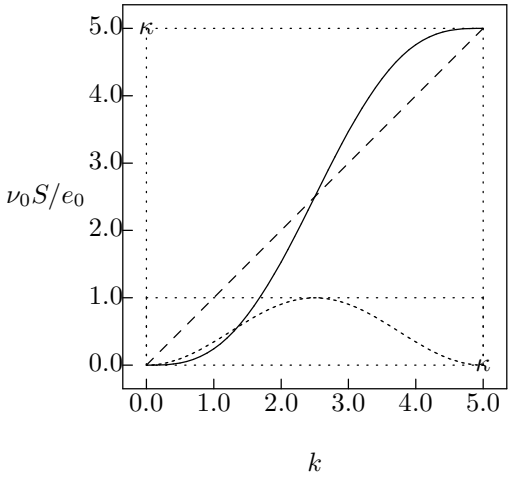

Figure 12: The relationship between kinetic energy and action as a function of time. The kinetic energy, shown as a dotted curve, defines a rectangle of size $\kappa \times 1$, where κ is the instruction's execution time, and divides the rectangle into two pieces of equal area. The case shown corresponds to $\kappa = 5$. The action, shown as a solid curve, is twice the area under the kinetic energy curve, and grows from zero before the instruction issues to the area of the rectangle when the instruction completes.

Figure 12 shows the relationship between kinetic energy and action for a typical instruction. The kinetic energy curve divides the $\kappa \times 1$ rectangle into two equal areas, and the numerical value of the action equals the area of the rectangle. In a system of measurement with the unit of work, $e_0 = 1$ flop, the unit of length, $l_0 = 1$ word, where a word is 8 bytes, and the unit of time, $t_0 = \nu_0^{-1}$ s, the inertial mass associated with the instruction has the value,

$$m = \left(\frac{8\kappa^2}{\pi^2 (l/l_0)^2} \right) m_0 \ , \tag{5.18}$$

where $m_0 = e_0/(\nu_0 l_0)^2$. For a machine with frequency, $\nu_0 = 1$ GHz, $m_0 = 10^{-18}$ flop $\cdot$ s^2/word^2.

5.2 A program's computational energy spectrum

Each instruction in a program generates kinetic energy as it executes, and the sum over all instructions is the program's energy spectrum. Consider a set of programs,

$$\mathcal{P} = \{P_1, P_2, \ldots, P_n\} \ , \tag{5.19}$$

each program consisting of a sequence of instructions,

$$P_i = \{I_i^1, I_i^2, \ldots, I_i^{n_i}\} \ , \tag{5.20}$$

with a possibly different number of instructions, n_i, for each program. Each instruction, I_i^j, issues at clock-tick k_i^j such that $0 < k_i^j < K_i$ where K_i is the total execution time for program P_i. Each instruction completes at clock-tick $k_i^j + \kappa_i^j$ where κ_i^j is the execution time for the instruction. The issue time and the completion time depend on the instruction sequence generated by the compiler and on the issue constraints imposed by the hardware.

For each instruction I_i^j in program P_i, define the dimensionless function,

$$T_i^j(k) = \begin{cases} 0 & k < k_i^j \\ T(k - k_i^j)/e_0 & k_i^j \le k \le k_i^j + \kappa_i^j \\ 0 & k > k_i^j + \kappa_i^j \end{cases} \ , \tag{5.21}$$

where $T(k)/e_0$ is defined by (5.14). The energy spectrum of the program is the sum over all its instructions,

$$T_i(k) = \sum_{j=1}^{n_i} T_i^j(k) \ . \tag{5.22}$$

At any given clock-tick, multiple instructions are likely to be executing, and definition (5.22) sums them all together.

Measurement of the energy spectrum requires two pieces of information, the issue time and completion time for each instruction as the program executes. This information may be difficult to obtain for a program running on actual hardware, but it is easy to obtain from an instruction-level simulation of the hardware. As a specific example, consider the simple program, shown in box (5.23), that computes the scalar product of two vectors.

```
#if defined(__GNUC__)
  #include <ppc_intrinsics.h>
#endif
int main(){
   int i, n=100;
   float s=0, x[n],y[n];
   for(i=0;i<n;i++) {x[i] = 1.0; y[i] = 1.0;}
   (void)__mfspr(1023);
   for(i=0;i<n;i++) {s += x[i]*y[i];}
   (void)__mfspr(1023);
}
```
(5.23)

The `amber` tool produces trace files for codes running on an Apple PowerPC G5 processor also known as the IBM PowerPc 970 processor [14]. The illegal instruction `__mfspr(1023)`, inserted twice into the code, signals the `amber` tool [13] to start a trace file at the first occurrence and to stop the trace at the second occurrence of the illegal instruction. From this trace file, the `simg5` simulator [2, 48] produces a detailed instruction trace with complete information at all stages of execution. In particular, it contains the clock-tick when each instruction issues and the clock-tick when each instruction completes. The instructions generated by the compiler between the two illegal instructions depend on the optimization level specified at compile-time.

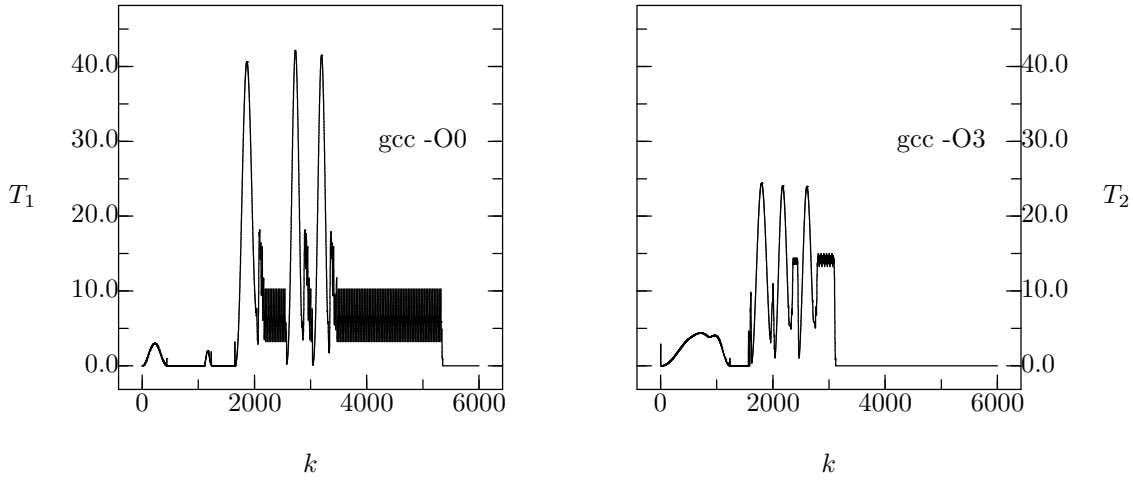

Figure 13: The kinetic energy spectra for program 5.23 compiled with different optimization levels. On the left, the code was compiled with the lowest optimization level, `gcc -O0`. On the right, the code was compiled with the highest optimization level, `gcc -O3`. The high spikes in the spectrum correspond to cache misses as discussed in the text. The thick, black parts of the spectrum correspond to floating-point computations. The optimized code not only executes in less time but also generates less action.

Figure 13 shows two energy spectra for program (5.23). They resemble molecular spectra familiar to chemists and physicists and contain diagnostic information related to the program's performance. On the left, the code is compiled with the lowest optimization level, `gcc -O0`; on the right, the code is compiled with the highest optimization level, `gcc -O3`. They both show an initial startup phase, of about 1500 clock-ticks, caused by the trap into the `amber` tool when it encounters the illegal instruction.

The two spectra are qualitatively the same, because they each represent the same computation, but quantitatively different, because the instructions generated by the compiler are different for the two optimization levels. The usual comparison focuses on the time axis where it is clear that the optimized code executes in less time, $K_2 = 3120$ clock-ticks compared with $K_1 = 5355$ clock-ticks for the unoptimized code.

But the energy axis contains more important information than the time axis. It shows why the program executes as it does and what causes delays in execution. For example, after the initial startup phase, each spectrum has three sharp spikes. A quick look at an excerpt from the instruction trace (5.24) for the optimized version of the program shows that these spikes are caused by cache misses.

Instruction	opcode	k_{issue}	k_{complete}	κ
47	bc	1600	1620	20
48	lfs	1602	1621	19
49	lfs	1602	1621	19
50	addi	1602	1621	19
51	addi	1602	1621	19
52	fmadd	1618	1626	8
53	bc	1603	1626	23
54	lfs	1606	1948	342
55	lfs	1606	1948	342
56	addi	1606	1948	342
57	addi	1606	1948	342
58	fmadd	1946	1954	8
59	bc	1607	1954	347

(5.24)

Between branch instructions `bc`, which mark each trip through the `for loop`, the optimized code executes two load instructions `lfs` to fetch the operands, two integer instructions `addi` to increment the address to the next pair of operands, and one fused multiply-add instruction `fmadd` to compute the result. At instructions 48 and 49, the addresses used by the load instructions hit in cache, and the instructions complete in 19 clock-ticks. At instructions 54 and 55, however, the addresses miss cache, and the instructions complete in 342 clock-ticks. The four instructions, two loads and two integer adds, issue as a group and complete as a group. A delay caused by a cache miss, therefore delays all four instructions, and the floating-point instruction cannot issue until the operands have arrived from memory. Once data resides in the cache, each loop executes in about 20 clock-ticks represented by the solid black areas of each spectrum.

The height of the spectrum measures approximately how many instructions are in flight at any given clock-tick. Each instruction contributes one unit of energy to the spectrum as shown in Figure 11. Since the spectrum is the sum of all the contributions, its height measures the number of instructions active at each clock-tick. The issue queues on the PowerPC are quite large so that theoretically as many as 215 instructions may be active at the same time [48]. Just considering that the two floating-point queues can hold 20 instructions, the two integer/load/store queues can hold 36 instructions, and the branch queue can hold 12 instructions, there might be as many as 68 instructions in flight at any given clock-tick. The spectrum for the unoptimized code, on the left side of Figure 13, shows spikes about 40 units high indicating that 40 instructions are in flight at that time. The optimized code, on the right side, only has about 25 instructions in flight at the peak of its spectrum. The extract from the instruction trace (5.24) shows at least twelve instructions active between clock-tick 1600 and clock-tick 1954.

Figure 14 shows the energy spectrum decomposed, something like a seismogram or an electrocardiogram, into its components for the four instructions that account for 97% of the action generated

by the optimized code.

Instruction	$S_2(K_2)$	Fraction
addi	19822	0.43
lfs	14852	0.32
bc	9027	0.20
fmadd	800	0.02
sum	44501	0.97
all	45743	1.00

(5.25)

The spikes in the full spectrum do indeed correspond to spikes for the load instruction and the instructions that depend on it. The spectrum for the multiply-add instruction shows where the floating-point work occurs in the program. It contributes almost nothing to the spectrum.

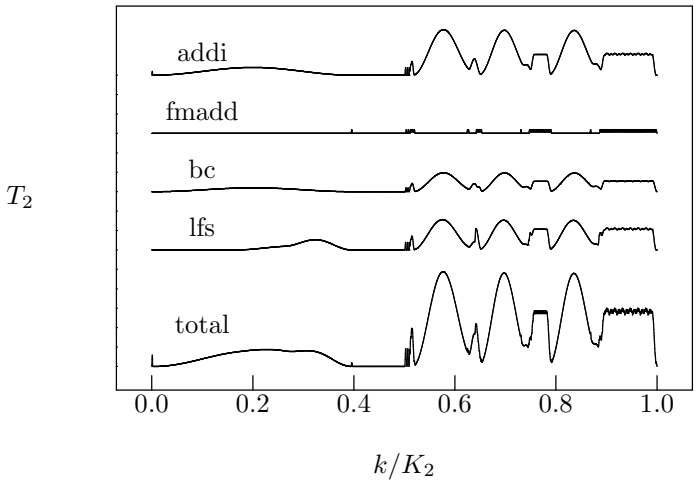

Figure 14: The kinetic energy spectrum decomposed into component parts for each instruction. The detailed behavior of the full spectrum at the bottom is clearly explained in terms of cache misses that occur periodically as the code executes. The contribution from the floating-point instruction adds little to the overall magnitude of the spectrum. Splitting it out separately clearly shows where floating-point work occurs during execution and how it is affected by memory delays.

Not surprisingly, the behavior of the code is determined by the memory hierarchy not by the floating-point units. The energy spectrum is a graphic depiction of where and why the code spends its time during execution and reveals where attention should be paid to further optimization of the code and perhaps to further changes to the hardware.

5.3 The metric space and the action norm

The action norm turns the set of programs (5.19) into a normed metric space [35, 44, 47]. The norm measures the size of each program and induces a distance function between two programs [28, 32].

To define the norm, multiply (5.21) by two and integrate to obtain the dimensionless function,

$$S_i^j(k) = \begin{cases} 0 & k < k_i^j \\ \nu_0 S(k - k_i^j)/e_0 & k_i^j \le k \le k_i^j + \kappa_i^j \\ \kappa_i^j & k > k_i^j + \kappa_i^j \end{cases} , \tag{5.26}$$

for each instruction I_i^j in program P_i. The function $\nu_0 S(k)/e_0$ is defined by (5.16), and $\kappa_i^j = \nu_0 S(\kappa_i^j)/e_0$ from (5.17) is the total action generated during execution. This function is zero until the instruction issues, grows according to (5.16) during execution, and retains the constant value (5.17) after the instruction completes. The total action generated by the program is the sum over all instructions,

$$S_i(k) = \sum_{j=1}^{n_i} S_i^j(k) \ . \tag{5.27}$$

From (5.17) and (5.26), the total action generated by program P_i at the end of execution has the value,

$$S_i(K_i) = \sum_{j=1}^{n_i} \kappa_i^j \ . \tag{5.28}$$

Programs that execute instructions with short completion times generate less action than programs that execute instructions with long completion times.

Since the total execution time K_i for program P_i is at most the sum of the execution times for all its instructions, the action generated by the program provides an upper bound,

$$K_i \le S_i(K_i) \ , \tag{5.29}$$

for the execution time in clock-ticks. If the instructions execute strictly sequentially, equality holds. If they overlap, the inequality holds. Optimization of generated code, therefore, is an exercise in scheduling instructions in such a way to minimize the total execution time to a value below the total action generated.

The next step in the definition of the norm is to shift the time scale to the dimensionless variable [28, p. 113],

$$z = 1 + \frac{k - K_i}{K_{\max}} \ , \tag{5.30}$$

where $K_{\max}$ is the maximum execution time over the set of programs,

$$K_{\max} = \max_i K_i \ . \tag{5.31}$$

In this new time scale, all programs finish at $z = 1$. Programs with maximum execution time, $K_i = K_{\max}$, begin execution at $z = 0$ while all other programs begin execution later at $z = 1 - K_i/K_{\max}$.

For each program P_i, define the function,

$$s_i(z) = \begin{cases} 0 & 0 \le z \le 1 - K_i/K_{\max} \\ S_i(K_{\max}(z-1) + K_i)/s_* & 1 - K_i/K_{\max} \le z \le 1 \end{cases} , \tag{5.32}$$

evaluating each program's action function at time $k = K_{\max}(z-1) + K_i$ and scaling it by the maximum action generated,

$$s_* = \max_i S_i(K_i) \ , \tag{5.33}$$

by any program in the set. The function $s_i(z)$ remains zero until the program begins execution at $z = 1 - K_i/K_{\max}$ and reaches its maximum value when the program finishes execution at $z = 1$. The program that generates the most action has the value $s_i(1) = 1$ when it finishes and all others have lower values less than one.

With these definitions, the set (5.19) of programs $\mathcal{P}$ becomes a normed metric space [28, 32] with the norm,

$$\|P_i\| = \int_0^1 |s_i(z)|dz \ , \tag{5.34}$$

that measures a program's size. A program's size, in this context, is not the number of lines of code or the number of instructions executed or the amount of memory used. Its size measures the action generated as it moves through computational phase space. The distance function,

$$\|P_i - P_j\| = \int_0^1 |s_i(z) - s_j(z)|dz \ , \tag{5.35}$$

is the natural distance function induced by the norm. It measures how far apart programs are from each other in terms of the difference in their paths through phase space.

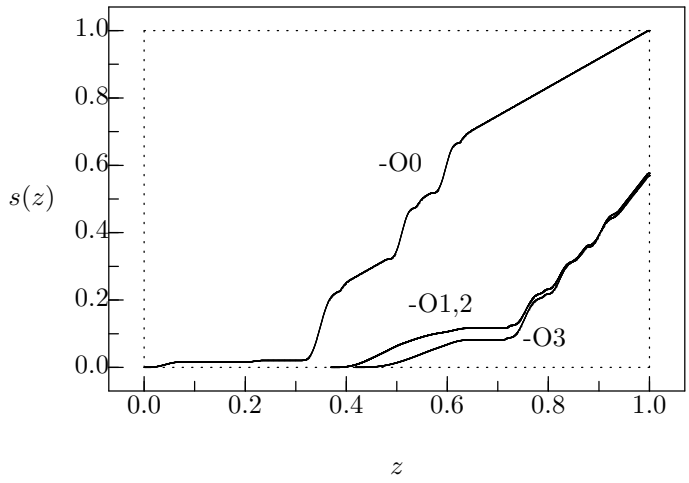

Figure 15: Computational action for four different optimization levels. The curves marked `-O0` and `-O3` correspond to the area under the kinetic energy spectra shown in Figure 13. The curves marked `-O1,2` correspond to intermediate levels of optimization whose spectra have been omitted. Almost all the performance improvement has been gained already at the first level of optimization. The highest optimization level yields a program with the least size that runs in the least time.

Figure 15 shows the action functions for program (5.23) compiled with four different optimization levels. Turning off all optimization, `gcc -O0`, yields a program that generates the most action. It corresponds to the spectrum on the left side of Figure 13. The highest level of optimization, `gcc`

`-O3`, yields the least action. It corresponds to the spectrum on the right side of Figure 13. The highest level of optimization is only marginally better than the two intermediate levels, `gcc -O1` or `gcc -O2`.

As already suggested by the difference in height of the energy spectra shown in Figure 13, the unoptimized code uses many more instructions than the optimized code. This difference is again reflected in the action curves shown in Figure 15 where the total action generated by the unoptimized code, $\nu_0 S_i(K_i = 5355)/e_0 = 79363$, is almost twice as big as that generated by the optimized code, $\nu_0 S_j(K_j = 3120)/e_0 = 45743$. The most important difference in the optimized code is that the compiler uses the fused multiply-add instruction, `fmadd`, which it did not use with optimization turned off.

The norm of each program is the area under its action curve in Figure 15. The program generated by the highest level of optimization has the smallest norm in this set of four programs and it has the smallest execution time. The distance between codes is the area between the action curves. The three programs with optimization levels -O1,2,3 are essentially the same, but all three are quite different from the unoptimized program.

6 Productivity metrics

In a very similar way, I have applied the idea of computational least action to the definition of new productivity metrics for software development [45, 47].

6.1 A metric space for programmers

For this case, rather than considering a set of *programs*, consider a set of *programmers*,

$$\mathcal{P} = \{P^1, P^2, \dots P^N\} \ , \tag{6.1}$$

assigned to a specific project. Let $t \geq 0$ represent time measured from the beginning of the project at time $t = 0$. Let T^i be the time spent on the project by programmer P^i and let

$$\mathcal{T}^i = [0, T^i] \tag{6.2}$$

be the corresponding time interval.

Programmers spend their time doing different things at different times during the project. To reflect this changing activity, divide each time interval $\mathcal{T}^i$ into subintervals,

$$\mathcal{T}^i_j = [t^i_{j-1}, t^i_j] \ , \qquad j = 1, n^i \ . \tag{6.3}$$

Each programmer starts at

$$t^i_0 = 0 \ , \tag{6.4}$$

and finishes at

$$t^i_{n^i} = T^i \ . \tag{6.5}$$

The number of intervals n^i is different for each programmer, and the total time spent T^i is different for each programmer. The width of each time interval is

$$\sigma^i_j = t^i_j - t^i_{j-1} \ , \tag{6.6}$$

and the activity performed in each interval is different for each programmer.

In each time interval $\mathcal{T}_j^i$, programmer P^i is involved in some activity that contributes some work, $W_j^i(t)$, toward finishing the project. Some activities advance the project more than others. For each activity, the power function of equation (6.27) is the derivative of the work function,

$$\rho_j^i(t) = \frac{dW_j^i}{dt} \ , \tag{6.7}$$

the rate of work production for programmer P^i in time interval $\mathcal{T}_j^i$.

For simplicity, assume that the power function ρ_j^i is constant in each interval so that

$$W_j^i(t) = \rho_j^i \int_{t_{j-1}^i}^{t} ds \ , \tag{6.8}$$

and hence work accumulates linearly in each interval,

$$W_j^i(t) = \rho_j^i (t - t_{j-1}^i) \ . \tag{6.9}$$

At the end of each time interval, the work accumulated over that interval is

$$W_j^i(t_j^i) = \rho_j^i \sigma_j^i \tag{6.10}$$

using the width of the interval from equation (6.6).

As time increases from one interval to the next, work accumulates at different rates at different times. At time $t \in \mathcal{T}_k^i$ the total accumulated work,

$$W^i(t) = W_k^i(t) + \sum_{j=1}^{k-1} \rho_j^i \sigma_j^i \ , \tag{6.11}$$

is the sum of the work done during all the intervals preceding interval $\mathcal{T}_k^i$ plus the additional work done so far in interval $\mathcal{T}_k^i$.

The action generated in each time interval is the integral,

$$S_j^i(t) = 2 \int_{t_{j-1}^i}^{t} W_j^i(s) ds \ , \tag{6.12}$$

with the factor of two inserted for convenience. Substitute the work function from equation (6.9) into the integral and evaluating the integral to find

$$S_j^i(t) = \rho_j^i (t - t_{j-1}^i)^2 \ . \tag{6.13}$$

At the end of each interval, the action accumulated over that interval is

$$S_j^i(t_j^i) = \rho_j^i (\sigma_j^i)^2 \ . \tag{6.14}$$

The total accumulated action in interval $\mathcal{T}_k^i$ at time t is the sum,

$$S^i(t) = S_k^i(t) + \sum_{j=1}^{k-1} \rho_j^i (\sigma_j^i)^2 \ . \tag{6.15}$$

The set of programmers $\mathcal{P}$ becomes a metric space [28] by defining a distance function based on the difference in how each programmer generates action during the project. This function should be a dimensionless function of a dimensionless variable such that the distance between programmers is a pure number. It should also measure a programmer's individual contribution to the project.

First define a set of units. The unit of time, T, is the maximum time spent by any programmer in the set,

$$T = \max_i(T^i) \ . \tag{6.16}$$

The unit of power is ρ and the unit of action is

$$\hat{S} = \rho T^2 \ . \tag{6.17}$$

To put each programmer onto the same time scale, define the dimensionless time variable,

$$z = 1 + (t - T^i)/T \ . \tag{6.18}$$

Define a dimensionless action function $s^i(z)$ in interval $\mathcal{T}_k^i$ from the sum in equation (6.15) evaluated at time

$$t = Tz + T^i - T \tag{6.19}$$

and scaled by the unit of action $\hat{S}$,

$$s^i(z) = \frac{1}{\hat{S}} \cdot \left[S_k^i(Tz + T^i - T) + \sum_{j=1}^{k-1} \rho_j^i (\sigma_j^i)^2 \right] \ . \tag{6.20}$$

The dimensionless time variable z spans the interval

$$1 - T^i/T \leq z \leq 1 \ , \tag{6.21}$$

and the first time interval for each programmer shifts to a new starting point,

$$z = 1 - T^i/T \ . \tag{6.22}$$

At this value of z, from definition (6.18), the time, $t = 0$, corresponds to the left end of the first interval where the action is zero. Extend the action function continuously to $z = 0$ by defining

$$s^i(z) = 0 \ , \qquad 0 \leq z \leq 1 - T^i/T \ . \tag{6.23}$$

In the variable z, every programmer ends activity at the same time,

$$z = 1 \ . \tag{6.24}$$

The programmer spending the longest time spans the whole interval from $z = 0$ to $z = 1$.

With these definitions, define the size of each programmer's contribution as the L_1-norm,

$$\|P^i\| = \int_0^1 |s^i(z)| dz \ , \tag{6.25}$$

and the distance between two programmers,

$$\|P^i - P^j\| = \int_0^1 |s^i(z) - s^j(z)|dz \ . \tag{6.26}$$

This new metric space for productivity in software development [47] is an extension of my earlier work on computational action metrics [37]. It will change the way we think of productivity in a fundamental way. This work is also related to two companion papers [44, 46] that shows how to define a similar metric space based on computational action for programs as they execute.

6.2 Data collection

We collected data from students as they worked on a programming assignment for a graduate-level course on Grid Computing at the University of Maryland [12]. The assignment was to implement Conway's Game of Life [19] to run in parallel on a Beowulf Linux cluster [4]. The students used the MPI library [15] to implement the parallel program.

We also collected data in collaboration with the DARPA HPCS Project. At the University of California, San Diego, Alan Snavely included Co-Array Fortran in his graduate-level course in parallel programming. I presented a Co-Array Fortran tutorial to his class with a live, interactive demonstration of how to use Co-Array Fortran on the Cray-X1 back at the AHPCRC in Minneapolis. The students were assigned a Sharks and Fishes problem using both Co-Array Fortran and MPI.

Vic Basili's students from the University of Maryland measured their programming effort to compare their productivity using the two programming models. The students reported very favorable impressions of their experiences with Co-Array Fortran compared with their experiences with MPI. We performed the same experiment with John Gilbert's class at the University of California, Santa Barbara with similar results.

We collected data by instrumenting the compiler. Each time a student compiled a program, we asked two questions. First, how long have you been working before the compilation? A blank response indicated that they had been working continuously since the previous compilation. Second, what kind of work were you doing? The student selected the kind of work from a list of seven activities, which are listed in the first column of Table 1.

Table 1: Activities and Power Ratings

Activity	Power Rating
Tuning	0.9
Parallelizing	0.7
Functionality	0.6
Learning	0.5
Compile-Time Error	0.2
Run-Time Error	0.2
Other	0.1

The instrumented compiler recorded the responses along with a time stamp indicating when the compilation occurred. From this data, we computed a set of time intervals for each student along with the activity associated with that interval.

A fundamental problem in trying to define a productivity metric in software development is the definition of work. Each kind of activity in each time interval corresponds to some work that advances the student toward the solution of a problem. Some activities advance the student more quickly than others, that is, they produce work at a higher. It is sufficient, for our purposes, to know the rate at which work accumulates, the power rating, without defining the actual unit of work itself. One unit of work can be converted to any other unit of work through a suitable conversion factor. The work associated with each activity can be converted into whatever unit of work we want without changing our results.

The important quantity for our analysis is the unit of power, ρ, which we define as the maximum rate at which any programmer can perform work to finish the project. In each interval of time, each programmer, denoted by superscript i, performs some activity, denoted by subscript j, at some fraction of peak power,

$$\rho_j^i = \alpha_j^i \rho \ . \tag{6.27}$$

The dimensionless parameters $0 \leq \alpha_j^i \leq 1$ characterize the behavior of each programmer. Table 1, in its second column, shows the power ratings, α_j^i, assigned to each activity. We have given all programmers the same power rating for the same activity although we could, with more information, assign different ratings to each one.

These power ratings are the input parameters to our model. Their values are purely subjective at this point, and we claim no profound meaning to them. They are dimensionless quantities that represent the fraction of peak power for each activity.

Figure 16 shows the action functions defined by equation (6.20) for the set of ten programmers we considered. Each programmer is marked by a symbol at the beginning and end of the corresponding interval in the dimensionless time variable z. The size of each programmer's contribution is the area under the action function. The distance between programmers is the area under the absolute difference between action functions.

Table 2: Individual contributions in milli-action units, $\rho T^2 \times 10^{-3}$. The values on the diagonal are the individual contributions from equation (6.25). The values below the diagonal are the distances between contributions from equation (6.26).

1	1.2									
2	0.8	1.9								
3	5.3	4.7	6.6							
4	3.3	3.3	5.4	3.0						
5	1.4	0.9	4.4	2.8	2.2					
6	3.6	3.4	4.8	0.8	2.8	3.5				
7	1.6	1.6	5.2	1.7	1.2	2.2	1.4			
8	6.3	5.9	5.9	3.6	5.1	3.0	5.1	6.5		
9	5.2	5.0	6.2	2.1	4.4	1.6	3.7	1.6	5.0	
10	1.3	1.8	6.3	2.7	2.0	3.3	1.1	6.3	4.8	0.3
	1	2	3	4	5	6	7	8	9	10

We can approximate the area under each curve by the area of the triangle determined by the end points of each curve [37]. Table 2 lists the values obtain this way in milli-action units, $\rho T^2 \times 10^{-3}$.

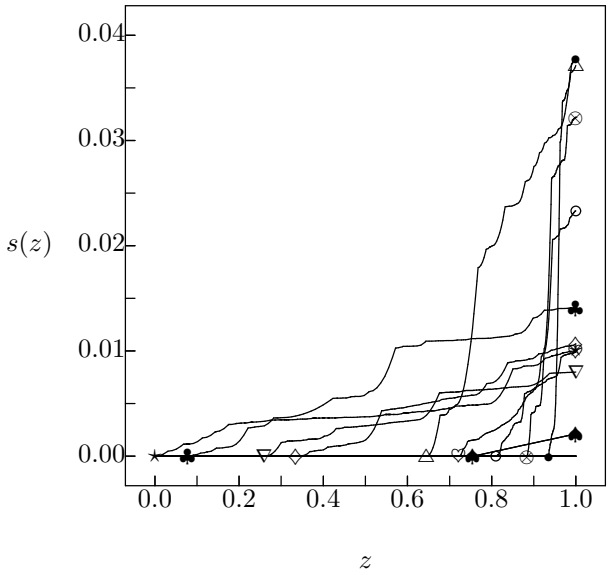

Figure 16: Action as a function of time for ten different programmers as a function of time. Each programmer is assigned a symbol that marks the beginning and ending of each curve. Time has been scaled so that the unit of time equals the longest time spent by any programmer in the set. The time for other members of the set are shifted to the right so that each programmer starts work at a different time but ends work at the same time.

7 The Principle of Computational Least Action

The two metric spaces defined in Section 5.3 for a set of programs and in Section 6.1 for a set of programmers, suggests the following principle of least action.

The Principle of Computational Least Action. *Given a set of programs (programmers), $\mathcal{P} = \{P_1, P_2, \ldots, P_n\}$, each solving the same problem, find a program (programmer) $P_* \in \mathcal{P}$ such that*

$$\|P_*\| = \min_{P_i \in \mathcal{P}} \|P_i\| \ , \tag{7.1}$$

where the norm $\|P_i\|$ measures the action for P_i.

The minimum may, of course, not be unique. There is more than one way to optimize a program and more than one good way to produce the code. For example, Figure 8 shows the action curves for two programmers, number three and number eight in Table 2, whose contributions are approximately equal. Although the area under the two curves is about the same, indicating that the two programmers contributed about the same amount to the project, the way they contributed is quite different. One programmer took a long but steady approach while the other took a short but steep approach. The quantitative measure of the difference between the two approaches is the area

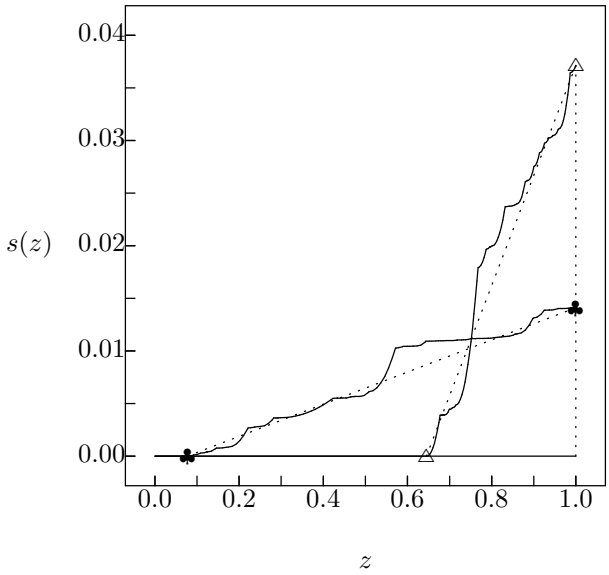

Figure 17: Action as a function of time for programmers three and eight from Table 2. The area under each curve is approximated by the triangle determined by the end points of each curve. The two programmers contributed about the same to the project, the area under the two curves is about the same, but they worked in two quite different ways to the same end.

between the two curves, which, from the corresponding entry in the table, equals 5.9 milli-action units.

This principle provides an entirely new way of measuring program performance and programmer productivity. It is the most important outcome of this project.

8 Publications resulting from the project

8.1 Peer-reviewed publications

Robert W. Numrich, Computer Performance Analysis and the Pi Theorem, *Journal of the ACM*, under review, 2008.

Robert W. Numrich, Computational Forces in the SAGE Benchmark, *Journal of Parallel and Distributed Computing*, under review, 2008.

Robert W. Numrich, Computational Forces in the Linpack Benchmark, *Journal of Parallel and Distributed Computing*, in press, 2008.

Robert W. Numrich, A Metric Space for Computer Programs and The Principle of Computational Least Action, *The Journal of Supercomputing*, 43(3):281-298, 2008.

Robert W. Numrich, The computational energy spectrum of a program as it executes, *The Journal of Supercomputing*, under review, 2008.

Robert W. Numrich, Dimensional analysis applied to a parallel QR algorithm, *Parallel Processing and Applied Mathematics: Proceedings of the Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM07)*, September 9-12, 2007, Gdansk, Poland, Springer Lecture Notes in Computer Science, LNCS 4967, 148-157, 2008.

Robert W. Numrich, Computational force: A unifying concept for scalability analysis, Parallel Computing: Architectures, Algorithms and Applications, Proceedings of the International Conference ParCo 2007, John von Neumann Institute for Computing (NIC) and Jülich Supercomputing Centre, 107-112, 2008.

Robert W. Numrich, A note on scaling the Linpack benchmark, *Journal of Parallel and Distributed Computing*, 67(4): 491-498, 2007.

Robert W. Numrich, Lorin Hochstein, Victor Basili, A Metric Space for Productivity Measurement in Software Development, *Proceedings SE-HPCS'05, Second International Workshop on Software Engineering for High Performance Computing System Applications*, St. Louis, Missouri, May 15, 2005.

Robert W. Numrich, Performance Metrics Based on Computational Action, *International Journal of High Performance Computing Applications*, 18(4): 449-458, 2004.

8.2 Conferences, tutorials and workshops

Robert W. Numrich, Computational Forces in the Linpack Benchmark, *13th SIAM Conference on Parallel Processing for Scientific Computing (PP08)*, Atlanta, GA, March 12, 2008.

Robert W. Numrich, The Principle of Computational Least Action, Louisiana State University, CCT Seminar, January 17, 2007.

Robert W. Numrich, Computer Performance Analysis and the Pi Theorem, Louisiana State University, CCT Seminar, January 18, 2007.

Robert W. Numrich, A New Scaling Formula for the Linpack Benchmark, *SIAM Conference on Computational Science and Engineering*, Costa Mesa, CA, February 19-23, 2007.

Robert W. Numrich, The Principle of Computational Least Action, Albert Einstein Institute, Potsdam, Germany, September 14, 2007.

Robert W. Numrich, What Does the Pi Theorem Tell Us about Computer Performance Analysis?, High Productivity Computing Systems Productivity Team Meeting, Marina del Rey, CA, January 10-11, 2006.

Robert W. Numrich, A Metric Space for Productivity in Software Development, High Productivity Computing Systems Productivity Team Meeting, Marina del Rey, CA, January 11-13, 2005.

9 Summary and future work

Two very important new results have come from this project. The first important result is the identification of the balance of computational forces as the key to understanding program performance. This breakthrough followed directly from the definition of a consistent system of measurement for computer performance analysis and the application of the methods of dimensional analysis. So far, I have applied it to a few parallel algorithms starting with very simple ones to help understand the basic procedure and progressing to more complicated full parallel applications. This procedure can be applied to any parallel application, and I intend to extend the procedure to as many parallel applications as possible.

The second important result is the statement of the Principle of Computational Least Action. It provides an entirely new theoretical framework for thinking about performance and productivity. I intend to apply it to more complicated programs and to perform detailed analysis of real programs as they execute. I intend to automate the process of producing energy spectra and the process of displaying the decomposed energy components, much like a musical score, such as the one in Figure 14.

It would be interesting to prove theorems based on the Principle of Computational Least Action. For example, is the computational action a minimum if and only if the computational time is a minimum? Or is it a minimum if and only if the computational work is a minimum? I intend to investigate these questions.

References

[1] Anant Agarwal, Mark Horowitz, and John Hennessy. An Analytical Cache Model. *ACM Transactions on Computer Systems*, 7(2):184–215, May 1989.

[2] IBM alphaWorks. Full-System Simulator for IBM PowerPC 970. http://www.alphaworks.ibm.com/tech/systemsim970, 2006.

[3] V. I. Arnold. *Mathematical Methods of Classical Mechanics*. Springer-Verlag, New York, 2nd edition, 1989.

[4] D. J. Becker, T. Sterling, D. Savarese, J. E. Dorband, U. A. Ranawake, and C. V. Packer. Beowulf: A parallel workstation for scientific computation. In *Proceedings of the 1995 International Conference on Parallel Processing (ICPP)*, 1995.

[5] Raymond T. Birge. On electric and magnetic units and dimensions. *American Physics Teacher*, 2(2):41–48, May 1934.

[6] Raymond T. Birge. On the establishment of fundamental and derived units, with special reference to electric units. Part I. *American Physics Teacher*, 3:102–109, 1935.

[7] Raymond T. Birge. On the establishment of fundamental and derived units, with special reference to electric units. Part II. *American Physics Teacher*, 3:171–179, 1935.

[8] Garrett Birkhoff. *Hydrodynamics: A Study in Logic, Fact and Similitude*. Princeton University Press, 2nd edition, 1960.

[9] Louis Brand. The Pi Theorem of Dimensional Analysis. *Arch. Rat. Mech. Anal.*, 1:35–45, 1957.

[10] P. W. Bridgman. *Dimensional Analysis*. Yale University Press, New Haven, 2nd edition, 1931.

[11] E. Buckingham. On physically similar systems: Illustrations of the use of dimensional equations. *Physical Review*, 4:345–376, 1914.

[12] J. Carver, S. Asgari, V. R. Basili, L. Hochstein, J. Hollingsworth, F. Shull, and M. V. Zelkowitz. Studying code development for high performance computing: The hpcs program. In *Workshop on High Productivity Computing, Edinburgh, Scotland*, pages 32–36, May 2004.

[13] Apple Developer Connection. CHUD Tools Manual Page for amber(1). http://developer.apple.com/documentation/Darwin/Reference/ManPages/man1/amber.1.html.

[14] Apple Developer Connection. Technical Note TN2087: PowerPC G5 Performance Primer. http://developer.apple.com/technotes/tn/tn2087.html, 2003.

[15] J.J. Dongarra, S.W. Otto, M. Snir, and D. Walker. A message-passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, 1996.

[16] S. Drobot. On the foundations of dimensional analysis. *Studia Mathematica*, 14:84–99, 1954.

[17] C. M. Focken. *Dimensional Methods and Their Applications*. Edward Arnold and Co., London, 1953.

[18] G. C. Fox, S. W. Otto, and A. J. G. Hey. Matrix algorithms on a hypercube I: Matrix multiplication. *Parallel Computing*, 4:17–31, 1987.

[19] M. Gardner. The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223:120–123, 1970.

[20] Herbert Goldstein. *Classical Mechanics*. Addison-Wesley, 1950.

[21] Bruce Greer and Greg Henry. High Performance Software on Intel Pentium Pro Processors or Micro-Ops to TeraFLOPS. In *Proceedings of Supercomputing '97*, pages 1–13, 1997.

[22] A. Hartstein, V. Srinivasan, T.R. Puzak, and P.G. Emma. Cache Miss Behavior: Is It $\sqrt{2}$? In *Proceedings of the 3rd Conference on Computing Frontiers*, pages 313–320. ACM Press, New York, May 3-5 Ischia, Italy 2006.

[23] Greg Henry and Robert A. van de Geijn. Parallelizing the QR algorithm for the unsymmetric algebraic eigenvalue problem: myths and reality. *SIAM Journal on Scientific Computing*, 17(4):870–883, July 1996.

[24] Roger W. Hockney. *The Science of Computer Benchmarking*. Society for Industrial and Applied Mathematics (SIAM), Philadelphia, 1996.

[25] HPC Challenge Benchmark. http://icl.cs.utk.edu/hpcc/.

[26] John David Jackson. *Classical Electrodynamics*. John Wiley & Sons, 1962.

[27] 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 *Proceedings of Supercomputing 2001*, Denver, CO, November 2001.

[28] A. N. Kolmogorov and S. V. Fomin. *Introductory Real Analysis*. Dover, revised English edition, 1970.

[29] Cornelius Lanczos. *The Variational Principles of Mechanics*. University of Toronto Press, 4th edition, 1949.

[30] Douglas Miles. Compute intensity and the FFT. *Proceedings Supercomputing 1993*, pages 676–684, 1993.

[31] National Institute of Standards and Technology. Prefixes for binary multiples. http://physics.nist.gov/cuu/Units/binary.html.

[32] Arch W. Naylor and George R. Sell. *Linear Operator Theory in Engineering and Science*. Holt, Rinehart and Winston, Inc., New York, 1979.

[33] Robert W. Numrich. Cray-2 memory organization and interprocessor memory contention. In Carl Meyer and R. J. Plemmons, editors, *Linear Algebra, Markov Chains, and Queueing Models*, volume 48 of *The IMA Volumes in Mathematics and Its Applications*, pages 267–294. Springer-Verlag, 1992.

[34] Robert W. Numrich. Memory contention for shared memory vector multiprocessors. In *Proceedings of Supercomputing '92*, pages 316–325. IEEE Computer Society, 1992.

[35] Robert W. Numrich. The computational action norm and the principle of computational least action. In *Proceedings of the Eighth SIAM Conference on Parallel Processing for Scientific Computing*, Philadelphia, March 1997. SIAM Activity Group on Supercomputing, Society for Industrial and Applied Mathematics.

[36] Robert W. Numrich. Computational Force, Mass, and Energy. *International Journal of Modern Physics C*, 8(3):437–457, June 1997.

[37] Robert W. Numrich. Performance metrics based on computational action. *International Journal of High Performance Computing Applications*, 18(4):449–458, 2004.

[38] Robert W. Numrich. Parallel numerical algorithms based on tensor notation and Co-Array Fortran syntax. *Parallel Computing*, 31:588–607, 2005.

[39] Robert W. Numrich. A note on scaling the Linpack benchmark. *Journal of Parallel and Distributed Computing*, 67(4):491–498, April 2007.

[40] Robert W. Numrich. Computational force: A unifying concept for scalability analysis. In Christian Bischof, Martin Bücker, Paul Gibbon, Gerhard Joubert, Thomas Lippert, Bernd Mohr, and Frans Peters, editors, *Parallel Computing: Architectures, Algorithms and Applications, Proceedings of the International Conference ParCo 2007*, pages 107–112. John von Neumann Institute for Computing (NIC) and Jülich Supercomputing Centre, 2007.

[41] Robert W. Numrich. Computational forces in the Linpack benchmark. In press, Journal of Parallel and Distributed Computing, March 2008.

[42] Robert W. Numrich. Computational Forces in the SAGE Benchmark. under review, January 2008.

[43] Robert W. Numrich. Dimensional analysis applied to a parallel QR algorithm. In *Parallel Processing and Applied Mathematics: Proceedings of the Seventh International Conference on Parallel Processing and Applied Mathematics (PPAM07)*, pages 148–157, September 9-12, 2007, Gdansk, Poland, 2008. Springer Lecture Notes in Computer Science, LNCS 4967.

[44] Robert W. Numrich. A metric space for computer programs and the principle of computational least action. *The Journal of Supercomputing*, 43(3):281–298, March 2008.

[45] Robert W. Numrich. A metric space for productivity in software development. In *High Productivity Computing Systems Productivity Team Meeting*, Marina del Rey, CA, January 11-13, 2005.

[46] Robert W. Numrich. The computational energy spectrum of a program as it executes. The Journal of Supercomputing, Under review, 2008.

[47] Robert W. Numrich, Lorin Hochstein, and Victor Basili. A metric space for productivity measurement in software development. In *Proceedings SE-HPCS'05, Second International Workshop on Software Engineering for High Performance Computing System Applications*, St. Louis, Missouri, May 15, 2005.

[48] Amit Singh. *Mac OS X Internals: A Systems Approach*. Chapter 3. Addison Wesley Professional, 2006.

[49] Jaswinder Pal Singh, Harold S. Stone, and Dominique Thiébaut. A Model of Workloads and Its Use in Miss-Rate Prediction for Fully Associative Caches. *IEEE Transactions on Computers*, 41(7):811–825, July 1992.

[50] G. W. Stewart. Communication and matrix computations on large message passing systems. *Parallel Computing*, 16:27–40, 1990.

[51] Barry N. Taylor. Guide for the Use of the International System of Units (SI). Special publication 811, National Institute of Standards and Technology, 1995.

[52] Barry N. Taylor. The International System of Units (SI). Special publication 330, National Institute of Standards and Technology, 2001.

[53] Dominique Thiébaut. On the fractal dimension of computer programs and its application to the prediction of the cache miss ratio. *IEEE Transactions on Computers*, 38(7):1012–1026, 1989.

[54] Dominique Thiébaut, Joel L. Wolf, and Harold S. Stone. Synthetic Traces for Trace-Driven Simulation of Cache Memories. *IEEE Transactions on Computers*, 41(4):388–410, April 1992.

[55] T. J. Willmore. *An Introduction to Differential Geometry*. English Language Book Society. Oxford University Press, 1959.