# A FRAMEWORK FOR MEASURING SUPERCOMPUTER PRODUCTIVITY

**Marc Snir**[1]
**David A. Bader**[2]

## Abstract

We propose a framework for measuring the productivity of high performance computing (HPC) systems, based on common economic definitions of productivity and on utility theory. We discuss how these definitions can capture essential aspects of HPC systems, such as the importance of time-to-solution and the trade-off between programming time and execution time. Finally, we outline a research program that would lead to the definition of effective productivity metrics for HPC that fit within the proposed framework.

Key words: Productivity, High-performance computing, performance metrics, benchmarks

## Acknowledgments

This work is supported in part by DARPA Contracts NBCHC-02-0056 and NBCH30390004. MS is also supported in part by DOE Grant DE-FG02-03ER25560. DAB is also supported in part by NSF Grants CAREER ACI-00-93039, ITR EIA-01-21377, Biocomplexity DEB-01-20709, and ITR EF/BIO 03-31654. The content of this paper evolved as the result of discussions that involved James C. Browne, Brad Chamberlain, Peter Kogge, Ralph Johnson, John McCalpin, Rami Melhem, David Padua, Allan Porterfield, Peter Santhanam, Burton Smith, Thomas Sterling, and Jeffrey Vetter. We thank the reviewers for their useful comments.

*The International Journal of High Performance Computing Applications*, Volume 18, No. 4, Winter 2004, pp 417–432
DOI: 10.1177/1094342004048535


## 1 Introduction

Productivity is an important subject of economic study, for obvious reasons. At the microeconomic level, an increase in productivity results in an increase in firm profits. At the macroeconomic level, an increase in productivity boosts the gross domestic product (GDP). An important concern is to ensure that productivity is properly measured, as improper measures will lead to suboptimal investment and policy decisions, at the micro and the macro levels.

The recent Defense Advanced Research Projects Agency (DARPA) High Productivity Computing Systems (HPCS) initiative (http://www.darpa.mil/ipto/programs/hpcs/) has raised the issue of proper productivity measurements, in the context of high performance computing (HPC). The motivation is the same as in the general economic realm: proper productivity measures will lead to better investments and better policy decisions in HPC. The HPC world has been obsessively focused on simple performance metrics, such as peak floating-point performance, or performance on the Linpack benchmark (TOP500 Supercomputer Sites, http://www.top500.org/). It would seem that these performance measures have often been used as proxies for total productivity measures, resulting in gross distortions in investment decisions.

In this paper we propose a framework for measuring the productivity of HPC systems. The basic framework is introduced in Section 2 and is discussed in Section 3. A possible approach for defining productivity metrics is introduced and discussed in Section 4. We conclude with a brief discussion.

## 2 Basics

Productivity is usually defined, in economics, as the amount of output per unit of input. Thus, the labor productivity of workers in the car industry can be measured as the number of cars produced per work hour (or, inversely, as the number of work hours needed to produce a car). The labor productivity of the US economy is measured as the ratio between the GDP and worker hours in the US. In this definition, one input measure (hours worked) is compared to one output measure (cars produced or GDP).

It is important to realize, in the first example, that the productivity of the car workers depends not only on their skills or motivation but also on the equipment available in car factories, and on the type of car manufactured. By the

[1] COMPUTER SCIENCE, UNIVERSITY OF ILLINOIS AT URBANA CHAMPAIGN, USA

[2] ELECTRICAL AND COMPUTER ENGINEERING, UNIVERSITY OF NEW MEXICO, ALBUQUERQUE, NM, USA

measure of work hours per car, workers in a heavily automated (more capital intensive) plant are likely to be more productive than workers in a less automated (less capital intensive) plant. Also, workers that produce Rolls-Royce cars are likely to be less productive than workers that produce Saturn cars, if productivity is measured in the number of cars.

The last bias can be avoided, to a large extent, if instead of counting car units one counts units of car value, i.e. the monetary value of cars produced in one work hour. In some contexts it may be useful to also measure the units of input in terms of their cost, e.g. by counting salary expenses, rather than hours of work. This is especially important for inputs that are heterogeneous and have no obvious common measure. If we do so, then productivity becomes a unit-less measure, namely the ratio between value produced and value of one of the inputs consumed to produce this value.

In some cases, it is useful to consider more than one input. Total productivity measures attempt to consider the sum of all inputs together. In order to obtain a meaningful measure, the various inputs need to be properly weighted. One obvious choice is to weight them by their cost, in which case productivity becomes the ratio between the units of output (or value of output) and cost of inputs.

The same approach can be used to measure the productivity of a factor or a set of factors, e.g. the productivity of a car plant, the productivity of a production line, the productivity of information technology in the car industry, etc.

Suppose that we are to apply the same approach to define the productivity of supercomputing systems. The productivity of a supercomputer would be defined as the total units of output produced by this system, divided by its cost. We are faced with an immediate problem: how do we measure the output of a supercomputer? Clearly, simple measures such as the number of operations performed or the number of jobs run will not do. As we do not have an efficient market for supercomputing results, we cannot use output market value, either.

Utility theory, as axiomatized by von Neumann and Morgenstern (1947) (see also Fishburn, 1970), provides a way out. Utility theory is an attempt to infer subjective value, or utility, from choices. Utility theory can be used in decision making under risk or uncertainty. Utility theory can be assumed to be descriptive, i.e. used to describe how people make choices. Alternatively, it can be assumed to be prescriptive or normative, i.e. used to describe how people should make choices under a rational model of decision making. This paper espouses this latter view.

Formally, one assumes a set $X$ of outcomes and a relation $\succ$ that expresses preferences of some rational agent over outcomes or lotteries of outcomes. A "lottery" is a finite subset of outcomes $\{x_1, ..., x_k\} \subset X$, with associated probabilities $p_1, ..., p_k$; $\sum p_i = 1$. We denote by $\Delta(X)$ the set of lotteries over outcomes in $X$. Note that $\Delta(X)$ is a convex set: $x, y \in \Delta(X)$ are lotteries, and $0 \le \alpha \le 1$, then $\alpha x + (1 - \alpha)y$ is a lottery in $\Delta(X)$.

The preference relation is assumed to fulfill the following axioms.

1. $\succ$ is *complete*, i.e. either $x \succ y$ or $y \succ x$ for all $x, y, z \in \Delta(X)$.
2. $\succ$ is *transitive*, i.e. if $x \succ y$ and $y \succ z$, then $x \succ z$ for all $x, y \in \Delta(X)$ .
3. *Archimedean axiom*: if $x \succ y \succ z$ then there exist $\alpha$ and $\beta$, $0 < \alpha < \beta < 1$, such that $y \succ \alpha x + (1 - \alpha)z$ and $\beta x + (1 - \beta)z \succ y$. This is a weak continuity axiom for lotteries.
4. *Independence Axiom*: $x > y$ if and only if $\alpha x + (1 - \alpha)z \succ \alpha y + (1 - \alpha)z$. This axiom asserts that preferences between $x$ and $y$ are unaffected if we combine both with another outcome in the same way.

These axioms are satisfied if and only if there is a real-valued utility function $U$ such that

1. $U$ *represents* $\succ$, i.e. $\forall x, y \in \Delta(x), x \succ y \Rightarrow U(x) > U(y)$;
2. $U$ is *affine*, i.e. $\forall x, y \in \Delta(X)$, $\alpha \in [0, 1]$, $U(\alpha x + (1 - \alpha)y) = \alpha U(x) + (1 - \alpha)U(y)$.

Furthermore, $U$ is unique, up to positive linear transformations. Thus, if "rational" preferences are expected to satisfy the four listed axioms, then any system of rational preferences can be represented by a utility function. The utility function is unit-less.

The utility function that represents the preferences of an agent can be estimated indirectly by assessing these preferences, e.g. by offering the agent a series of hypothetical choices. Clearly, this approach may not be very practical in an environment where there are multiple, complex outcomes. However, many of the results in this paper do not depend on the exact shape of the utility function.

One assumes that the total value of a supercomputer's output is defined by the (rational) preference of some agent (e.g. a lab director, or a program manager). The agent's preference can be represented by a utility function. We can now use the utility produced by the supercomputer as a measure of its output.

We shall define the productivity of a supercomputer system as the ratio between the utility of the output it computes to the cost of the system. This measure depends on the preferences of an agent, rather than being objectively defined. This is unavoidable: the value of the output of a supercomputer may vary from organization to organization; as long as there is no practical way to trade supercomputer outputs, there is no way to directly com-

pare the worth of a supercomputer for one organization to the worth of a supercomputer for another organization. (An indirect comparison can be made by ascertaining how much an organization is willing to pay for the machine.) In any case, much of what we shall say in this paper is fairly independent of the exact shape of the utility function. The units of productivity are utility divided by a monetary unit such as dollars.

The definition allows for some flexibility in the definition of a "system". This may include the machine only, or it may include all the expenses incurred in using the system, including capital and recurring cost for the building that hosts it, maintenance, software costs, etc. One definition is not intrinsically better than the other: The first definition makes sense when the only choice considered is whether to purchase a piece of hardware. The second definition makes sense when one considers long-term choices.

If one wishes to account for all costs involved in the use of a system, then one faces a cost allocation problem; many of the expenses incurred (such as replacement cost of the capital investment in a machine room) may be shared by several systems. It is not always straightforward to properly allocate a fraction of the cost to each system. Various solutions have been proposed to this problem (see, for example, Young, 1994). This issue of cost evaluation and cost allocation may be non-trivial, but is a standard accounting problem that occurs in the same form at any firm that uses information technology, and is addressed by many accounting textbooks. We shall not pursue this matter further.

To the same extent that we can extend the definition of what a "system" is, we can restrict it to include only the resources made available to one application. Here, again, we face the problem of proper cost allocation (or of proper resource pricing), which we do not address in this paper.

The utility of an output is a function of its timeliness. A simulation that predicts the weather for Wednesday November 5, 2003 is valuable if completed on Tuesday November 4, or earlier; it is most likely worthless if completed on Thursday November 6. Thus, the utility of an output depends not only on what is computed, but also on when the computation is performed. Completion time is part of the output definition, and is an argument that influences utility.

Our discussion will focus on time-to-solution as a variable parameter that affects the utility of an output. The result of a computation may have other parameters that affect its utility, such as its accuracy, or its trustworthiness (i.e. the likelihood that the code used is a correct implementation of the algorithm it is supposed to embody). In some situations, time-to-solution may be fixed, but other parameters can be varied. Thus, one may fix the period of time dedicated to a weather simulation, but vary the accuracy of the simulation (grid resolution, accuracy of physics representation, number of scenarios simulated, etc.). The general framework can accommodate such variants.

## 3 Discussion of Basic Framework

To simplify the discussion, we consider in this section an idealized situation where a system $S$ will be purchased uniquely to develop from scratch and solve once a problem $P$. By "system", we mean the programming environment that is used to develop the code and the (hardware and software) system that is used to run it. There are multiple possible choices of systems to purchase, programming language to use for implementing the application, programming efforts to invest in coding and tuning the application, etc. What is the rationale for such choices?

We study the problem by assuming three independent variables: $T$, the time-to-solution; $S$, the system used; and $U$, the utility function that represents preferences on outcomes. Thus, given, $S$ and $T$, we minimize the cost of solving problem $P$ on system $S$ in time $T$. We denote the (minimal) cost of such a solution by $C(P, S, T)$. Each time-to-solution $T$ is associated with a utility $U = U(P, T)$ and a productivity

$$\Psi(P, S, T, U) = \frac{U(P, T)}{C(P, S, T)}.$$

Then, the ideal productivity of system $S$ for problem $P$ is defined to be

$$\Psi(P, S, U) = \max_T \Psi(P, S, T, U) = \max_T \frac{U(P, T)}{C(P, S, T)}.$$

$\Psi(P, S, U(\cdot\,,\cdot))$ is the (best possible) productivity for problem $P$ on system $S$, relative to utility function $U = U(P, T)$. Note that productivity normally indicates a measured ratio of outputs to inputs, in a given environment; ideal productivity is a measure of productivity that can be potentially obtained, assuming optimal decisions.

Our ultimate purpose is to be able to compare the "value" of vendor systems. We say that system $S_1$ is better than system $S_2$ for problem $P$ with respect to utility function $U$ if $\Psi(P, S_1, U(\cdot\,,\cdot)) > \Psi(P, S_2, U(\cdot\,,\cdot))$.

The formalism of this section reflects two obvious observations.

1. The ideal productivity of a system depends on the type of application that is run on this system; a collection of networked PCs may be the most productive system to look for evidence of extraterrestrial intelligence (especially when the compute time is freely donated; Anderson et al., 2002), but may not be very productive for simulating nuclear explosions (Advanced Simulation and Computing,

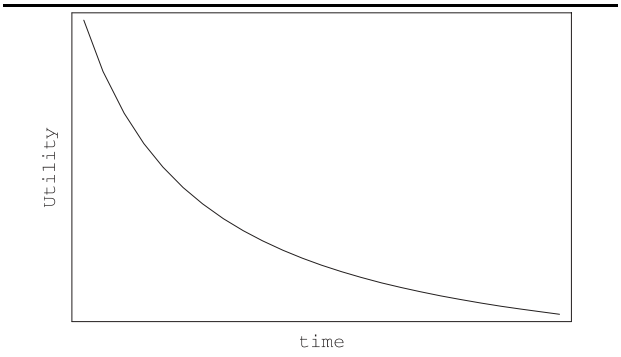


**Fig. 1 Utility decreases with time.**

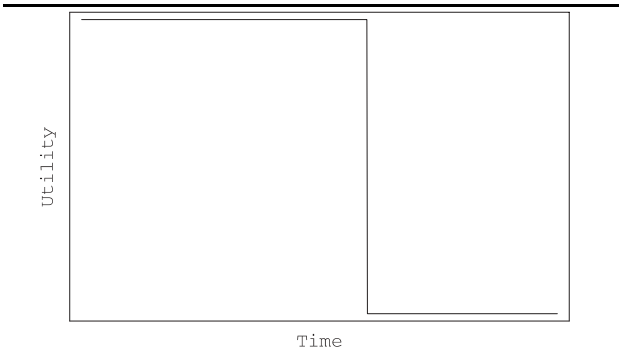


**Fig. 2 Deadline-driven utility.**

http://www.nnsa.doe.gov/asc/). For some applications, the ideal productivity is an open question; for example, protein folding using idle cycle-stealing from networked PCs (e.g. Stanford's Folding@Home project; Shirts and Pande, 2000) versus using innovative architectures (e.g. the IBM BlueGene project; Allen et al., 2001).

2. The productivity of a system depends on the time sensitivity of the result that is represented by the utility function $U$. If the result is as valuable if obtained 20 years from now as if obtained in a week, then there is no reason to buy an expensive supercomputer; one can just wait until the computation fits on a PC (or a game machine). On the other hand, if it is very important to complete the computation in a week, then an expensive high-end machine and a quick prototyping programming environment may prove the more productive system.

We can use the same approach to compare other alternatives, by suitably redefining what we mean by a "system". For example, assume that we want to compare two programming environments that are both supported on a vendor system, in order to decide which should be used to solve problem $P$. Define $S_1$ to be the system with the first programming environment and $S_2$ to be the system with the second programming environment, and proceed as before.

We proceed now to discuss more in detail costs and utilities.

## 3.1 UTILITY

Utility $U(P, T)$ is a decreasing function of time-to-solution $T$, as illustrated in Figure 1, because there is no loss in acquiring information earlier (information does not deteriorate), and there may be some benefit. (A possible counterexample is a situation where the cost of storing the output of a computation until it can be acted upon is high. However, in cases of interest to us, the cost of storing the result of a computation is insignificant compared to the cost of the computation itself.) In many environments, the value of a solution may steeply decline over time: intelligence quickly loses its value; weather must be simulated accurately in time to provide a prediction, etc. At the other extreme, fundamental research is relatively time-insensitive, so that value declines moderately with time. This motivates two value–time curves. A deadline driven utility is defined by

$$U(P, T) = \begin{cases} u & \text{if} \quad T \leq \text{deadline} \\ 0 & \text{if} \quad T > \text{deadline} \end{cases}. \tag{1}$$

This yields the step-function curve illustrated in Figure 2. The curve illustrated in Figure 3 corresponds to fixed, time-independent utility, i.e.

$$U(P, T) = u. \tag{2}$$

The curve in Figure 4 illustrates a staircase function that approximates a commonly occurring situation in industry, or in national labs (see, for example, Peterson et al., 1989): the highest utility is associated with "near-interactive time" computations, i.e. a computation that takes less than a coffee break so that it can be part of an uninterrupted engineering development cycle. Above a certain time, computations tend to be postponed to overnight runs, so the next level represents the utility of a computation performed overnight, which can be part of a daily development cycle. The third threshold corresponds to the util-

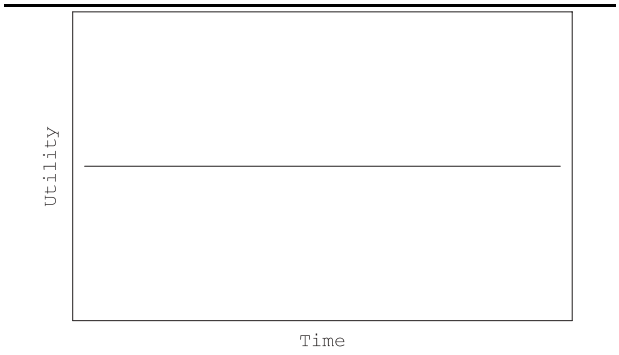


**Fig. 3 Time-independent utility.**

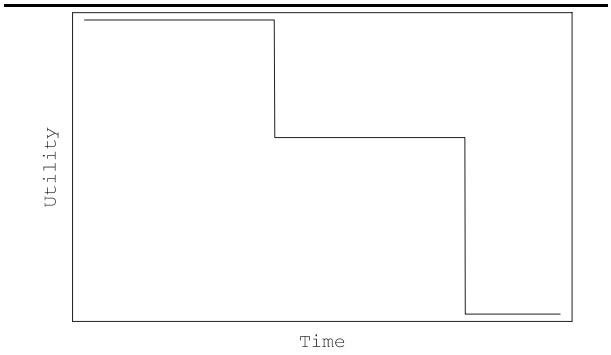


**Fig. 4 Staircase utility function.**

ity of computations that cannot be carried out overnight, and which tend to be done "once in a project lifetime".

## 3.2 COST

Cost can be decomposed into development cost $C_D$, and execution cost $C_E$. In a simplified model, development and execution are disjoint in time. Thus, $C(P, S, T) = C_D(P, S, T_D) + C_E(P, S, T_E)$, where $T_D$ is the development time, $T_E$ is the execution time, and $T = T_D + T_E$.

Execution cost is the cost of the system used for execution. This includes purchasing, deployment, and operation, cost of the system (hardware and software); cost of the maintenance and repair of the system; cost of the building needed to host the machine (if this is considered part of the system); cost of electrical power and other supplies; etc. The operation cost includes the time to train operators, insert and extract information and run the program (response time), and handle special requests from users. Some of the costs may be shared (e.g. cost of machine room) and need to be allocated appropriately.

Development cost includes initial development activities such as requirements gathering, design and planning, code development, documentation, and testing. It also includes the cost of operating the software and maintaining it, the cost to make changes and updates, the cost of porting to new systems, the cost of recovering from its failures, and the cost of integrating other software with it. A more accurate measurement should include other factors such as future savings in labor costs in future software projects if the software is well designed and can be reused. In this first approximation, we do not take all these additional factors into account. Development cost also includes the cost of training programmers when a new language or computer system is used.

Software cost is often measured in terms of hours of labor (assuming uniform wages), while hardware cost is often measured in terms of machine time. An initial performance study may focus on these two measures only. However, even if one is willing to only use these two coarse measures, one needs to worry about the "exchange rate" between these two measures. The time spent in parallelizing and tuning a high performance code can represent a significant fraction of the total software development effort. Nevertheless, it is seldom the case that tuning stops only when no further performance improvement is possible. This implies an implicit trade-off between the extra programming effort needed to further reduce execution time and the savings in machine time resulting from this effort. To address this trade-off, we need to measure both compute time and programmer time using a common measure, namely a monetary unit such as dollars. It is then possible (in principle) to also address the difference between the different classes of workers with a wide range of skills that are usually involved in the development and deployment of a software system.

## 3.3 WHAT IS THE SHAPE OF THE CURVES?

As explained in Section 3.1, the utility function $U(P, T)$ is most likely a decreasing function $C(P, S, T)$ of time. One can similarly expect the cost function to be a decreasing function of time. This is so because one needs to increase development cost in order to reduce development time and one needs to increase execution cost in order to decrease execution time. In order to reduce development time, one may increase the number of programmers. However, an increase in the number of programmers assigned to a programming task does not usually result in a proportional

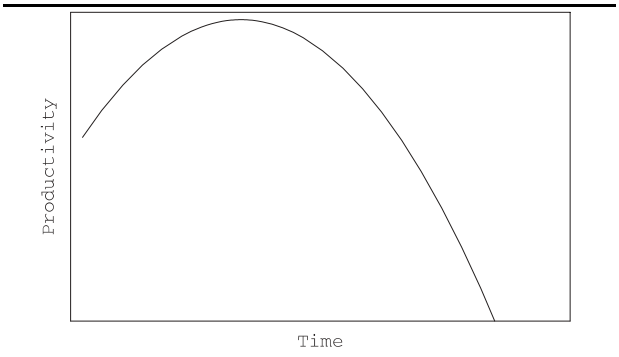


**Fig. 5 Productivity as a function of time-to-solution.**

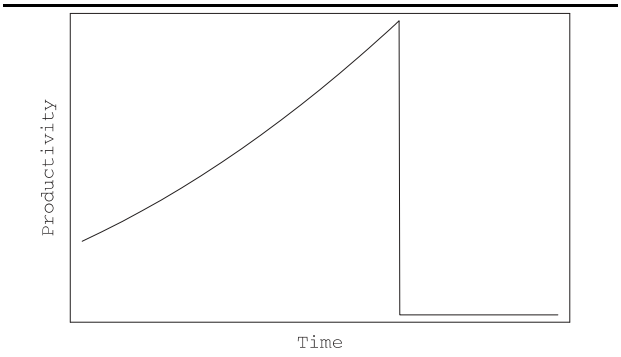


**Fig. 6 Productivity for deadline-driven utility function.**

decrease in development time (Brooks, 1975). Similarly, in order to reduce execution cost, one may use a machine with more processors. However, speedup is seldom linear in the number of processors. Thus, stretching the time-to-solution reduces cost (up to a point) but also reduces the utility of the result.

By composing the cost–time curve and the utility–time curve one obtains a productivity–time curve: $\Psi(P, S, T, U) = U(P, T)/C(P, S, T)$. Little can be said, a priori, about its shape: it depends on how fast cost decreases when one stretches the time-to-solution versus how fast the utility of the result decreases. In many cases we expect a unimodal curve as illustrated in Figure 5, with one optimal solution that gives us the ideal (optimal) productivity. At one end, if we stretch development time beyond the bare minimum possible then productivity increases, since the marginal cost for a reduction in time-to-solution is very steep, when we get close to the minimum. At the other end, if we stretch the time-to-solution too much then productivity decreases, since there are almost no further cost savings, while utility continues to drop.

Consider the following two extremes.

**Flat utility-time curve (equation (2)).** Productivity is inversely proportional to cost, and, hence is monotonically increasing:

$$\Psi(P, S, T, U) = \frac{U(P, T)}{C(P, S, T)} = \frac{u}{C(P, S, T)}.$$

If utility does not decrease over time, then the development and execution process with the slowest practical progress rate, which is the cheapest, will also be the most productive. There is no reason to ever use a parallel machine, or to have more than one application developer working on a project, if time is not an issue. Parallelism increases some costs in order to reduce time-to-solution. Its use is premised on the assumption of decreasing utility with time. This clearly indicates why one must take into account the fact that utility decreases over time.

**Deadline-driven utility–time curve (equation (1)).** Productivity is inversely proportional to cost and increases, up to the deadline, then immediately drops to zero:

$$\Psi(P, S, T, U) = \begin{cases} u/C(P, S, T) & \text{if} \quad T \le \text{deadline} \\ 0 & \text{if} \quad T > \text{deadline} \end{cases}.$$

Since cost is monotonically decreasing, we obtain a curve as illustrated in Figure 6. In this case, the most productive solution is the one that accelerates time-to-solution so as to meet the deadline, but no more. This ignores uncertainty and risk. In practice, uncertainty on the time needed to solve the problem will lead to a solution that provides a margin of safety from the deadline. We shall further revisit this issue in Section 3.5.

## 3.4 DUALITY

The discussion of the last sections assumed time $T$ to be the free parameter, and cost $C$ to be the dependent parameter. That is, one sets time-to-solution, and spends the amount needed in order to achieve that deadline. Instead, one can set the budget, and look at time-to-solution as dependent on budget. We define, accordingly, $T(P, S, C)$ to be the minimal time needed to solve problem $P$ on system $S$ with cost $C$. It is easy to see that

$$\Psi(P, S, U) = \max_c \frac{U(P, T(P, S, C))}{C}.$$

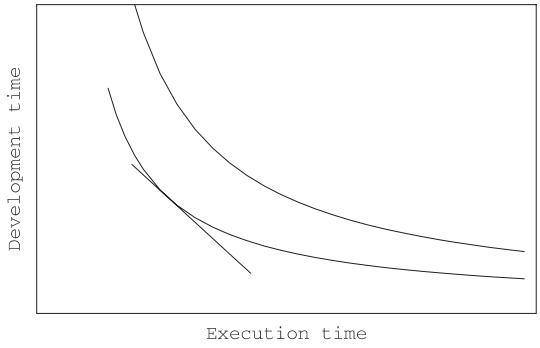


**Fig. 7 Constant cost curves.**

Using this formulation, productivity is maximized by considering, for each possible cost, the least time-to-solution achievable for this cost, and hence the best productivity achieved with this cost; then picking that cost that optimizes productivity. Thus, if one understands what time-to-solution can be achieved for each cost, one has the information needed to compute productivity, for any utility function. This formulation has the advantage of being closer to the way decisions are made in most firms and organizations: budget is allocated and completion time is a result of this allocation.

**3.4.1 Minimizing Time-to-Solution.** As mentioned in Section 3.2, time-to-solution $T(P, S, C)$ includes development time $T_D$ and execution time $T_E$: $T = T_D + T_E$. There is a trade-off between the two, as added development time can lead to reduced execution time: more time spent tuning and parallelizing an application leads to less running time. This is especially true for large-scale parallel systems, where added development time spent in parallelizing an application can reduce execution time by orders of magnitude. The same observation applies to applications that have to be executed in sequence many times: added development time can significantly reduce execution time.

Assume a fixed cost $C$ for development and execution. A fixed application can be developed and executed using different combinations of development time and execution time. These combinations will be described by a curve similar to the curves in Figure 7. Total time will be minimized by one (or more) points on this curve, the point where marginal increase in development time is equal to the marginal decrease in execution time. This optimum point defines $T(P, S, C)$. The optimum is achieved at point where the asymptote to the curve has a 135° slope. Indeed, we seek the minimum of $T = T_D + T_E$, under the constraint $C_D(P, S, T_D) + C_E(P, S, T_E) = C$. This is achieved when

$$\frac{\partial C_D}{\partial T_D} = \frac{\partial C_E}{\partial T_E}, \text{ or } \frac{\partial T_E}{\partial T_D} = -1.$$

The discussion in this section assumes that the cost functions are differentiable.

## 3.5 Uncertainty and Risk

The discussion in the previous sections assumed that the time-to-solution achieved for a given cost (or, equivalently, the cost of achieving a given time-to-solution) are perfectly known. In practice, there is some uncertainty about the time-to-solution. One can model this by assuming that $T(P, S, C)$ the time-to-solution function, is stochastic, i.e. $T(P, S, C)$ defines a probability distribution on the solution time, given the problem, system and cost. We now define

$$\Psi(P, S, U) = \max_c \frac{E(U(P, T(P, S, C)))}{C}.$$

In this model, for each cost $C$, we compute the ratio of the expected utility achieved over that cost. The (ideal) productivity is obtained by maximizing over $C$.

We analyze a concrete example, in order to illustrate these definitions. Assume a deadline driven utility:

$$U(P, T) = \begin{cases} 1 & \text{if} \quad T \le d \\ 0 & \text{if} \quad T > d \end{cases}.$$

Assume, first, that completion time is related to cost as:

$$T(P, S, C) = 1/C.$$

Then

$$\Psi(P, S, U) = 1/d.$$

Optimal decision making will stretch completion time $T(P, S, C)$ to match the deadline $d$, thus achieving utility 1, while minimizing cost to equal $C = 1 / d$. This is the result derived in Section 3.3.

Assume now that completion time is stochastic: the completion time is uniformly distributed in the interval $[(1 - \delta) / C, (1 + \delta) / C]$, where $0 < \delta < 1$. Then completion time has mean $1 / C$, and variance $\sigma^2 = \frac{1}{3}(\delta / C)^2$. Then

$$E(U(P, T)) = Pr(T \le d)$$
$$= \begin{cases} 0 & \text{if} \quad d < (1 - \delta)/C \\ \dfrac{d - (1 - \delta)/C}{2\delta} & \text{if} \quad (1 - \delta)/C \le d \le (1 + \delta)/C \\ 1 & \text{if} \quad d > ((1 + \delta)/C). \end{cases}$$

If $\delta < 1 / 3$ then the maximum of $E(U) / C$ is obtained when $C = (1 + \delta) / d$. The mean completion time then

equals $1 / C = d / (1 + \delta)$. Thus, the larger the variance, the smaller the expected completion time that maximizes productivity, and the smaller the productivity. A larger uncertainty on completion time forces one to a more conservative schedule, thus reducing the resulting utility. In this example, productivity is optimized by a schedule that totally avoids the risk of missing the deadline. In general, the optimal solution may be obtained for a schedule that entails some risk of missing the deadline.

A more robust analysis should account for the fact that uncertainty about time-to-completion decreases as a project progresses. Rather than a single decision point, one should think of a continuous decision process where, at any point in time, one picks the branch that maximizes expected productivity, given current knowledge.

## 3.6 RANKING SYSTEMS

It is tempting to expect that a study such as ours will provide one figure of merit for systems or, at the least, a linear ordering. This is unlikely to happen.

**Definition**: System $S_1$ dominates system $S_2$ ($S_1 \succ S_2$) if $\Psi(P, S_1, U) \geq \Psi(P, S_2, U)$ for any problem $P$ and utility function $U$.

If $S_1$ dominates $S_2$, then is clearly preferable to $S_2$. This could occur, for example, when $S_1$ is identical to $S_2$, except that $S_1$ is cheaper, or has hardware that runs faster, or a compiler produces faster code, etc. However, in general, systems are not well ordered: $\Psi(P, S_1, U) > \Psi(P, S_2, U)$, for some problem $P$ but $\Psi(P', S_1, U) < \Psi(P', S_2, U)$ for some other problem $P'$, or $\Psi(P, S_1, U') < \Psi(P, S_2, U')$ for some other utility function $U'$. For example, system $S_1$ is better for "scientific" applications, but system $S_2$ is better for "commercial" applications. Or, system $S_1$ is better when time-to-solution is paramount, but system $S_2$ is better for applications that are less time-sensitive. Thus, we should not expect one figure of merit for systems, but many such figures of merits. These do not provide a total ordering of systems from most productive to least, but rather indicate how productive a system is in a specific context, for a specific set of applications and specific time sensitivity. We discuss in the next section how one can define such figures of merits, i.e. productivity metrics.

# 4 Metrics and Predicting Time-to-Solution

The discussion in the previous sections implies that one can estimate the productivity of a system, if one can estimate the time-to-solution (or other critical features of a solution) achieved for various problems, given a certain investment of resources (including programmer time and machine time). Indeed, the time-to-solution determines the utility of the system and the resources invested determine the system cost; both together determine productivity. How can one undertake such a study?

An ideal framework for such a study is provided by the following approach. Each system $S$ is characterized by metrics $M_1(S)$, ..., $M_n(S)$. These metrics could include parameters that contribute directly to execution performance, such as the number of processors on the target system, processor speed, memory, etc.; parameters that relate to availability, such as MTBF; parameters that relate to programming performance, such as support for various languages, libraries and tools; and parameters that quantify the quality of this support.

The characteristics of such metrics are that they are

- (reasonably) easy to measure on existing systems and to predict for future systems;
- application independent.

Similarly, each problem $P$ is characterized by metrics $L_1(P)$, ..., $L_m(P)$. These application metrics could include static code parameters such as source lines of code (SLOCs); execution parameters such as number of instructions, instruction mix, and degrees of locality (e.g. spatial, temporal) found in the memory references; metrics to characterize the complexity of the software, and the complexity of the subject domain; metrics to characterize how well specified the problem is, and how likely the specification is to evolve; and so on. The characteristics of such metrics are that they are

- (reasonably) easy to measure on existing applications and to predict for future problems;
- system independent.

A model predicts time-to-solution as a function of the system and problem parameters:

$$T(P, S, C) \approx T(L_1(P), ..., L_m(P), M_1(S), ..., M_n(S), C).$$

The system metrics $M_1(S)$, ..., $M_n(S)$ will be productivity metrics for the system. For reasons explained in Section 3.6, we should not expect that one, or even a few, parameters will categorize a system: this is a high-dimensional space.

The existence of a model that can predict time-to-solution validates the system metrics: it shows that the metrics used are complete, i.e. have captured all essential features of the systems under discussion. Furthermore, the model provides a measure of the sensitivity of time-to-solution to one or another of the metrics. If the model is analytical, then $\frac{\partial T}{\partial M_i}$ measures how sensitive is time-to-solution to metric $M_i$, as a function of the system and the application under consideration.

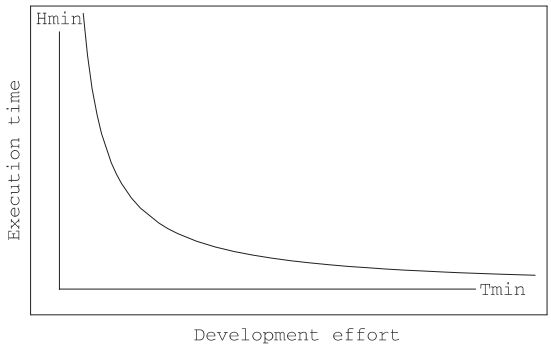


**Fig. 8 Execution time as function of development effort.**

Models are likely to be stochastic: one does not obtain a single predicted time-to-solution, but a distribution on time-to-solution. In fact, one is likely to do with even less; namely with some statistical information on the distribution of time-to-solution. The model validity is tested by measuring actual time-to-solution and then comparing it to predicted time-to-solution for multiple applications and multiple systems. The number of validating measurements should be large relative to the number of free parameters used in the model, so as to yield statistically meaningful estimates of the parameters and error margins.

We introduced cost as a model parameter in order to be able to compare different resources using a common yardstick. Researchers may prefer to use parameters that can be measured without referring to price lists or to procurement documents. Rather than cost, one can use parameters that, to first order, determine cost. The cost of a system is determined (to first approximation) by the system configuration (processor speed, number of processors, memory per processor, system bandwidths, interconnection, etc.); these parameters are part of the system metrics. The software development cost is determined (to first approximation) by the number of programmer hours invested in the development. Thus, a more convenient model may use

$$T_E(P, S, H) \approx T_E(L_1(P), ..., L_m(P), M_1(S), ..., M_n(S), H)$$

where $H$ is the number of hours invested in code development and $T_E$ is the execution time of the program as a function of the system (or system metrics), problem (or problem metrics), and hours of code development. We expect that the curve that depicts $T_E$ as a function of $H$ will be similar to the curve in Figure 8, with two asymptotes: there is a minimal development effort $H_{min}$ that is needed to get a running program, irrespective of performance; and there is a minimal execution time $T_{min}$ that will not be improved upon, irrespective of tuning efforts. A nearly L-shaped curve indicates rigidity: increasing development time does not significantly decrease the running time. In such a case the execution time can be approximated as

$$T_E = \begin{cases} \infty & \text{if} \quad H < H_{min} \\ T_{min} & \text{if} \quad H \geq H_{min} \end{cases}.$$

If so, then one can study development effort $H_{min}$ and execution performance $T_{min}$ separately; the first depends only on the software metrics of the problem and the systems, while the second depends only on the performance metrics of the problem and the system.

## 4.1 DISCUSSION

The general framework outlined in the previous section is not new. It has been used to study execution time (see, for example, Abandah and Davidson, 1998; Snavely et al., 2002), and it has been used, implicitly, in many software engineering studies, to predict development cost or maintenance cost of software as a function of various program metrics (see, for example, Boehm et al., 1995). We do not propose to repeat all the work that has been done in this area. However, there are good reasons to believe that new work is needed.

1. The development process for high performance scientific codes is fairly different from the development processes that have been traditionally studied in software engineering. Research is needed to develop software productivity models that fit this environment.
2. Development time and execution time have been traditionally studied as two independent problems. This approach is unlikely to work for HPC because execution time is strongly affected by investments in program development, and a significant fraction of program development is devoted to performance tuning. Note that much of the effort spent in the porting of codes from system to system is also a performance tuning effort related to different performance characteristics of different systems, rather than a correctness effort related to different semantics. Our formulation emphasizes this dependency.
3. The utility function imposes constraints on the development time allowable to communities that use HPC. For instance, one discipline may have problems with a deadline – driven utility, so users develop run-once "Kleenex" codes, while another

discipline may have problems with a nearly time-independent utility and use highly optimized software systems developed over many years.

There will be an inherent tension between the desire to have a short list of parameters and the desire to have accurate models. A full specification of performance aspects of large hardware systems may require many hundreds or thousands of parameters: feeds and speed of all components, sizes of all stores, detailed description of mechanisms and policies, etc. If we compress this information into half a dozen or so numbers, then we clearly lose information. A model with a tractable number of parameters will not be able to fully explain the performance of some (many?) applications.

The appropriate number of metrics to use will depend on the proposed use of these metrics and models. One may wish to use such metrics and models so as to predict with good accuracy the performance of any application on a given system. In such a case, the system metrics provide a useful tool for application design. Such a goal is likely to require detailed models and long parameter lists. Our goal is more modest: we would like a set of metrics and a model that provides a statistically meaningful prediction of performance for any fixed system, and for large sets of applications. While the error on the estimation of programming time or execution time for any individual application can be significant, the average error over the entire application set is small. This is sufficient for providing a meaningful "measure of goodness" for systems, and is useful as a first rough estimate of the performance a system will provide in a given application domain.

## 4.2 PERFORMANCE-ORIENTED SYSTEM METRICS

For over ten years, the HPC community has mostly relied upon a single metric to rank systems, i.e. the maximum achievable rate for performing floating-point operations while solving a dense system of linear equations (e.g. Linpack). As a significant service to the community and with substantial value to the vendors and customers of high-end systems, Dongarra et al. release these rankings, known as the Top500 List, in a semi-annual report (http://www.top500.org/). While this metric has encouraged vendors to increase the fraction of peak performance attained for running this benchmark problem, the impact of these optimizations on the performance of other codes is not clear. For instance, the performance of integer codes (from problems using discrete mathematics, techniques from operations research, and emerging disciplines such as computational biology) may not correlate well with floating-point performance. In addition, codes without spatial or temporal locality may not achieve the same performance as Linpack, a regular code with a high degree of locality in its memory reference patterns.

In an ideal model, productivity metrics will span the entire range of factors that affect the performance of applications. If the code is memory-intensive, and with a high degree of spatial locality but no temporal locality, performance typically depends upon the system's maximum sustainable memory bandwidth (for instance, as measured by the Stream Benchmark; HPC Challenge Benchmark, http://icl.cs.utk.edu/hpcc/; McCalpin, 2004). For another code with low degrees of locality (essentially uniformly random accesses to memory locations) the RandomAccess benchmark (see http://icl.cs. utk.edu/hpcc/), which reports the maximum giga-updates-per-second (GUPS) rate for randomly updating entries a large table held in memory, may be a leading metric for accurately predicting performance. Thus, when seeking a necessary set of system metrics for good accuracy of the model, metrics should reveal bottlenecks for achieving higher application performance. In modern systems, in addition to the memory bandwidth, integer performance, and memory latency, the following factors are often identified as limiting the performance for at least some codes. These factors illustrate some important metrics but are by no means a necessary or sufficient list of system metrics:

- sustainable I/O bandwidth for large transfers (best case);
- sustainable I/O bandwidth for large transfers that do not allow locality;
- floating-point divide, square root, transcendental functions;
- bit twiddling operations (data motion and computation);
- global, massively concurrent atomic memory updates for irregular data structures, such as those used in unstructured meshes, e.g. graph coloring, maximal independent set.

The metrics are formally defined by microbenchmarks, i.e. experiments that measure the relevant metric. In many cases, the experiment consists of running (repeatedly) a specific code, and measuring its running time.

The metrics so far reveal performance characteristics usually dependent on a single hardware feature. Metrics may also include more complex operations that depend on multiple hardware features. These metrics often are referred to as "common critical kernels", and could include metrics for the rates of the following types of operations:

- scatter/gather;
- scans/reductions;

- finite elements, finite volumes, fast Fourier transforms, linear solvers, matrix multiply;
- list ranking: find the prefix-sums of a linked list;
- optimal pattern matching;
- database transactions;
- histogramming and integer sorting;
- LP solvers;
- graph theoretic solvers for problems such as traveling sales person, minimum spanning tree, max flow, and connected components;
- mesh partitioners.

Such kernels can be defined by a specific code, or can be defined as a paper and pencil task (or both).

The advantage of using metrics that measure specific hardware features is that it is relatively easy to predict their values for new systems, at system design time. Indeed, in many cases, these parameters are part of the system specification. The disadvantage is that it may be harder to predict the performance of a complex code using only these basic parameters.

It is harder to predict the performance of a kernel when hardware is not extant. On the other hand, it may be easier to predict the performance of full-blown codes from the performance of kernels. In many cases, the running time of a code is dominated by the running time of kernels or solvers that are very similar or identical to the common critical kernels. In such a case, the performance of a code is not improved significantly by additional tuning, once a base programming effort has occurred. The performance can be modeled as a linear combination of the performance of the common kernels, i.e.

$$T_E(S, P, H) \approx \begin{cases} \infty & \text{if} \quad H < H_{\min} \\ \alpha_1 M_1(S) + \ldots + \alpha_n M_n(S) = T_{min} & \text{if} \quad H \geq H_{\min} \end{cases}$$

where the weights $\alpha_1, \ldots, \alpha_n$ are problem parameters and $M_1, \ldots, M_n$ are system metrics defined by the running time of common kernels on the system. (To simplify the discussion, we assume a fixed problem size and a fixed machine size. In the general case, they both need to appear as system and problem metrics, respectively.)

## 4.3 PERFORMANCE-ORIENTED APPLICATION METRICS

In order to estimate the execution time of a program, the system metrics need to be combined with application metrics. Ideally, these application metrics should be defined from the program implementing an application, with no reference to parameters from the system executing the program. However, almost no high-level programming notation provides a "performance semantics" that supports such analysis. Notable exceptions are NESL (Blelloch, 1996; Blelloch and Greiner, 1996) and Cilk (Blumofe et al., 1996). NESL provides a compositional definition of the *Work* (number of operations) and the *Depth* (critical path length) of a program, together with an implementation strategy that runs NESL programs in time $O(Work / P + Depth)$ on $P$ processors. Cilk provides an expected execution time of a "fully strict" multithreaded computation on $P$ processors, each with an LRU cache of $C$ pages, in $O((Work + mM) / P + mCDepth)$ where *Work* and *Depth* are defined as before, $M$ is the number of cache misses (in a sequential computation), $C$ is the size of the cache, and $m$ is the time needed to service a miss.

Application metrics are more commonly defined by using an abstract machine and specifying a mapping of programs to that abstract machine. At the least, this abstract machine model is parametrized by the number of processors involved in the computation (JaJa, 1992; Reif, 1993); the model may include more parameters describing memory performance (Aggarwal et al., 1989; 1990) or interprocessor communication performance (Culler et al., 1993). In this approach, one does not define explicitly application metrics; these are defined implicitly via algorithmic analysis which derives a formula that relates input size and abstract model parameters to running time. The prediction of the running time on an actual machine is achieved by setting the parameters of the abstract machine to match the actual machine characteristics.

Algorithmic analysis is essential in the design of performing software. Furthermore, algorithmic analysis can account correctly for complex interdependences which have the effect that the running time of a program is not a s imple addition of the running time of components. Unfortunately, this approach also has many limitations, as follows.

- A simple model such as the PRAM model facilitates analysis, but ignores many key performance issues, such as locality, communication costs, bandwidth limitations, and synchronization. One can develop more complex models to reflect additional performance factors, but then analysis becomes progressively harder. Furthermore, there is no good methodology to figure out what is "good enough": there is no way of bounding the error that the model obtains when certain factors are ignored.
- A complete analysis of a real application code is impractical. Instead, one makes educated guesses about the small performance-critical code regions and analyzes these, ignoring a large fraction of the remaining code. However, which parts of a large code cause performance bottlenecks may change from system to system.

- The mapping of an algorithm expressed in a high-level language to a system is a complex process, involving decisions made by compilers, operating systems, and run-time environments. These are typically ignored in algorithmic analysis. Worse, since algorithms are designed for a simple execution model (fixed number of processors, all running at the same speed), any deviation from the simple execution model appears as an anomaly that may cause performance problems, not as a strategy that improves performance.
- For many problems (e.g. optimization solvers and problems with irregular data structures such as those that use sparse matrix and graph representations), it is not sufficient to characterize input by size; in general, it may not be possible to derive a simple closed-form formula for the performance of a code.

An alternative measurement based approach is to extract application metrics from measurements performed during application execution. Typical metrics may include the following (see, for example, Abandah and Davidson, 1998; Snavely et al., 2002):

- the number of operations executed by the program;
- the characterization of memory accesses that includes their number and their distribution;
- the characterization of internode communication that includes the number of messages and the amount of data communicated between nodes.

Such parameters can be combined with machine parameters to predict the total running time on a system other than that where measurements were taken. This presumes that one has correctly identified the key parameters, has the right model, and that the application parameters do not change as the code is ported. For example, a compiler can affect operation count or memory access pattern; the effect is assumed to be negligible, in this approach.

Finally, the two approaches can be combined in a hybrid method that develops a closed-form analytical formula for running time as a function of application and machine parameters, but uses measurements to estimate some of the application parameters. For example, the execution time for sequential code might be measured, while communication time is estimated from a model of communication performance as a function of application traffic pattern and system interconnect performance (see, for example, Kerbyson et al., 2001). The hybrid approach has the disadvantage that it cannot be as fully automated as the measurement-based approach; it also has the advantage that it often can derive a parametric application model that is applicable to a broader set of inputs.

So far, we have assumed that applications are described as a combination of low-level operations, such as additions, multiplications, memory accesses, etc. Alternatively, it may be possible to characterize applications as combinations of higher level components, such as the common critical kernels described in Section 4.2. This is likely to be valid when most of the computation time is actually spent in one of the listed kernels. However, the approach may still be valid even if this is not the case, provided that the set of kernels is rich enough and "spans" all of the applications under consideration. Then, each application can be represented as a weighted sum of kernels (running on inputs of the right size).

## 4.4 SOFTWARE PRODUCTIVITY

System metrics can be defined using micro benchmarks, i.e. carefully defined experiments that measure key features of the system. This approach has been frequently used to characterize hardware features (Hristea et al., 1997; Hey and Lancaster, 2000). The design of such performance micro benchmarks is derived from an understanding of the key performance bottlenecks of modern microprocessors. The experiments attempt to measure the impact of each important hardware feature in isolation. It is tempting to attempt the use of a similar approach for measuring software productivity: we would characterize the key subactivities in HPC software development and attempt to measure the impact of the development environment on each such subactivity. A first step in this direction is to understand the key steps in software development and how they might differ in HPC, as compared to commercial software development.

**4.4.1 Requirements/Specifications.** This phase defines the requirements for the software and the necessary specifications. In the commercial environments, the biggest challenge is to keep changes to the requirements at a minimum during the execution of the project. In HPC projects, the challenge may not be managing the changes to the specifications, but gathering of the specifications in adequate details for clarity of design and implementation. The crashing of the Mars Climate Orbiter on the surface of the planet due to the software calculations being made in the wrong units (Imperial versus metric) shows the importance of this step. Any errors in the requirements can easily lead to churns in the design causing major productivity impacts. Software engineering tooling in this phase has been very weak.

**4.4.2 Design.** This is the phase of a software project when high-level (and, if applicable, low-level) architecture of the software is defined from a set of requirements. Typically this is carried out by highly skilled software/application designers. Depending on the process enforced and the size of the development group, the design docu-

ments can be very formal to casual. In the commercial arena, the productivity during design can be increased by the use of design languages, such as the unified modeling language (UML), which allow substantial automated analysis of the design and generate some lower level artifacts for an easier implementation. In the HPC arena, such as calculations in nuclear physics experiments, simulating combat scenarios, weather predictions, protein folding, etc., design is not likely to be written a priori. Conceptual progress, model definition, and programming are all performed concurrently by a skilled researcher/technician (who may not be a trained as a professional software engineer) and hence some of the design aids in commercial software do not apply. Design and modeling end up being captured in the code implementation and one may have some design documentation only after the fact for maintenance or analysis purposes. Parallel design also poses specific complexity usually not found in commercial environments.

**4.4.3 Code Development and Debugging.** This is the phase during which a code is developed and debugged from specification or design. In the commercial arena, significant progress has been made in providing integrated development environments (IDEs) that can significantly increase the programmer productivity. Typical IDEs come with higher level components that can be easily used by the programmer for producing significant functionality in a very short time. Some examples in this area are the development of graphical user interfaces (GUIs), connection to databases, network socket connections, etc. These environments are also quite capable of providing language specific debugging tools for catching errors via static code analysis (e.g. unassigned variables, missing pointers, etc.) There are also documented examples of improved productivity as a result of the adoption of Java as the programming language instead of C++. In the HPC arena, languages and development environments are still not advanced enough to be able to provide similar productivity enhancements in the creation and debugging of the parallel code. However, this is an area where significant progress is possible, by adapting existing IDEs to the HPC arena, adding tools that deal with parallel correctness and performance, and by taking advantage of existing/emerging components for HPC (libraries and frameworks).

**4.4.4 Testing and Formal Verification.** Testing of commercial software can easily take 30–70% of the developing effort depending on the nature of the software and the reliability assurance required (RTI, 2002). Testing involves four key activities: test design, test generation, test execution, and debugging. A clear understanding of what the software is expected to do is essential for efficient and effective testing of the software. In the commercial arena, there has been considerable automation in test execution compared to the other three activities. In practice, testing is rarely exhaustive. Even though it is the most common way of doing software quality assurance, testing (due to the sampling nature) does not preclude residual software defects in the program. Formal verification of the design or code or model checking may be the only way to ensure some critical behavior of the software, but this does not scale for a wide range of behavior. However, in the HPC arena, testing is typically carried out as a part of coding. Unlike commercial applications, typically the calculations are limited in the number of outputs the programs can produce. Therefore, for mission critical software applications, formal verification of the program and/or model checking may indeed be a viable option. Here, too, there should be many opportunities for adapting commercial tools and practices to HPC.

**4.4.5 Code Tuning for Performance.** By definition HPC codes have high performance requirements that often justify a significant investment in performance tuning, beyond what is common in commercial code development. This problem is compounded by the difficulties of achieving good performance on large scalable systems and by the large variation among hardware systems. Thus, tuning is likely to be a more significant component of software development for HPC than for commercial codes.

**4.4.6 Code Evolving.** As many HPC codes are experimental in nature, we can expect these codes to evolve at a faster rate than commercial codes. This increases the importance of tools for automated testing, especially of tools that can test a new code version against a "golden version". In many HPC environments, there is a close interaction between the code developers and the code users (indeed, they will often be one and the same). This close interaction has many obvious advantages. It has one important disadvantage in that it leads to a less formal process for defect reporting and tracking, and hence often to an inferior capture of process information about software evolution. Hence, tools that facilitate such capture are very important.

## 4.5 Software Productivity Benchmarks

**4.5.1 Process Benchmarks.** Although there is an extensive literature on software metrics, and models that predict how hard it is to develop and maintain codes, as a function of code metrics, there is little, if any, that applies specifically to HPC. While there is a vigorous debate on the productivity of various programming models, e.g. message passing versus shared memory, the debate is

almost entirely based on anecdotal evidence, not measurement. Thus, we believe that it is imperative to initiate a research program that will produce experimental data to settle such arguments.

We envisage software productivity experiments which focus on a well-defined task that can be accomplished by a programmer of some experience within a reasonable time frame, e.g. developing code with 500–1500 SLOC for a well-defined scientific problem. The experiments will focus on one aspect of software development and maintenance and will compare two competing environments: e.g. OpenMP (Official OpenMP Specifications, http://www.openmp.org/specs/) versus MPI (MPI: A Message-Passing Interface Standard, http://www.mpi-forum.org/docs/mpi-11-html/mpi-report.html). Compact application benchmarks, such as the NAS application kernels (Bailey et al., 1995), could be used for that purpose. Examples of software development activities are listed below.

**Code development and debugging.** Develop and debug a code from specification. This experiment tests the quality of the programming model and of the programming environment used for coding and debugging.

**Code tuning.** Tune code until it achieves 50% (or 70%, etc.) of "best" performance. Best performance can be defined by the performance achieved by a vendor tuned version of the code. These experiments are very important as they allow us to study the trade-off between coding time and execution time. Code tuning can be defined to involve performance for a fixed number of processors, or performance for a range of processor numbers, thus becoming a "code scaling" study. Note that code tuning tests not only the programming model, but the quality of the implementation and of the performance tools.

**Code evolving.** This is the same experiment as for code development and tuning, except that one does not start from a specification, but from an existing code. One can test evolution that results from a specification change (e.g. change solver), or evolution that results from a system change (e.g. port shared memory code to message passing, or vice versa). In the latter case, the difficulty of the task depends both on the source model and on the target model.

The most obvious measure of the difficulty of such an activity is programming effort: how long does it take a programmer to achieve the desired result? For results to be meaningful, one needs to perform experiments in a controlled environment (e.g. students after an introductory course, experienced programmers), with large enough populations so as to derive meaningful statistics. The experiments should be performed in academia or by research labs, not by the vendors themselves.

**4.5.2 Product Benchmarks.** In addition to (or instead of) the process metric of programmer time, one can measure product metrics: metrics of the produced software. In the traditional commercial software development environment, there have been different attempts to measure software productivity and relate it to software metrics. Typically this is expressed as the ratio of what is being produced to the effort required to produce it. For example, the number of SLOC is a commonly used measure of the programmer output and then the number of lines of code per programmer month is the resulting productivity metric. This has the specific advantage that the measurement of the lines of code in a program can be easily automated. However, it suffers from the most common criticism that the same functionality can be delivered by different implementations with widely different lines of code count based on the style of the programmer, programming language, development environments using CASE tools, etc. This results in a serious possibility of manipulation of the measurement of the lines of code to increase the productivity value. Also, the number of lines of code does not quite capture the complexity or the specific aspects of functionality implemented by a programmer. Thus, as an alternative to the lines of code, some organizations have used function points as the measure of programmer output (Albrecht and Gaffney, 1983). Function points are defined explicitly in terms of the implemented function (number of inputs, outputs, etc). Unfortunately, due to some of the subjective elements of the functional definition, the counting of the function points cannot be automated (Verner et al., 1989). Also, some valid and real programming elements do not end up contributing to function points, and consequently this metric ends up undercounting implemented functionality. More recently, with the increasing popularity of software design using UML, there is more interest in considering some measurements based on the UML artifact, "use case". So far, this approach still suffers from the handicap that not all use cases are the same, and a more detailed characterization of a use case is necessary before serious software output measures can be defined. The literature is replete with many alternative measures for code "complexity" (e.g. cyclomatic complexity; McCabe and Watson, 1994). Of particular importance for HPC are more recently developed software metrics that seem more applicable to OO code (Tahvildari and Singh, 2000a). The measurement of many of those can be automated, but it is not always clear how well they predict programming effort. Of particular importance for our purpose is to develop and assess software metrics that capture the "parallel complexity of codes". Programmers have an intuitive notion of code that has low "parallel complexity" (simple communication and synchronization patterns), compared to code with high "parallel complexity" (complex communication and synchronization patterns). We expect more synchronization bugs in codes with high parallel complexity. It would be useful to provide a for-

mal definition to this intuitive notion and to correlate "parallel complexity" to debugging and tuning time.

We are left with no clear choice to adopt for measuring software productivity in the HPC environment from the traditional commercial software arena. Our problem is compounded by the fact that the development of HPC code has its own unique challenges (difficulty of coding parallel code, importance of performance tuning), and by the fact that we are interested in models that can be used to compare programming models or systems. This is not a focus of most work on software productivity. While the problem is challenging, we feel this still is a direction worth pursuing.

We can use product metric measurements as a "proxy" for the three activities we discussed in the previous section.

**Code development.** For example, we can assess the quality of a particular programming model by the complexity of the software needed to code a particular problem from a specification. This approach has been used to assess the impact of using pattern-based systems for parallel applications (Tahvildari and Singh, 2000b).

**Code tuning.** One can assess the difficulty of tuning for a particular system by comparing the code complexity of a "vanilla" implementation to the code complexity of an optimized implementation.

**Code evolution.** One can assess the difficulty of code evolution by measuring the complexity of the changes needed in a code to satisfy a new requirement.

## 5 Summary

In this paper we have outlined a framework for measuring the productivity of HPC systems and we have described possible experiments and research directions suggested by this framework. Some of this research is now starting, under the DARPA HPCS program (http://www.darpa.mil/ipto/programs/hpcs/). The problem of measuring productivity is a complex one; it has vexed economists over many decades, in much simpler contexts than that considered in this paper. Thus, we should not expect quick answers or complete solutions. However, the problem of measuring productivity is also fundamental to any rational resource allocation process in HPC. Thus, even partial progress in this area is better than nothing. We hope that the framework suggested in this paper will contribute to such progress.

## AUTHOR BIOGRAPHIES

*David A. Bader* is an Associate Professor and Regents' Lecturer in the Departments of Electrical and Computer Engineering and Computer Science at the University of New Mexico (UNM). He received his PhD in electrical engineering in 1996 from the University of Maryland, and was awarded a National Science Foundation (NSF) Postdoctoral Research Associateship in Experimental Computer Science before joining UNM in 1998. He is an NSF CAREER Award recipient, an investigator on several NSF awards, a distinguished speaker in the IEEE Computer Society Distinguished Visitors Program and Dr Bader serves on the Steering Committees of the IPDPS and HiPC conferences, and is the General Co-Chair for IPDPS (2004–2005), and Vice General Chair for HiPC (2002–2004). He has served on numerous conference program committees related to parallel processing, is an associate editor for the IEEE Trans on Parallel and Distributed Systems and the ACM Journal of Experimental Algorithmics, a Senior Member of the IEEE Computer Society, and a Member of the ACM. He has co-authored over 60 papers on parallel algorithms, combinatorial optimization, and computational biology and genomics.

*Marc Snir* is Michael Faiman and Saburo Muroga Professor and Head of the Department of Computer Science at the University of Illinois at Urbana-Champaign. Dr Snir's research interests include large-scale parallel systems, parallel computer architecture, and parallel programming. He received a PhD in mathematics from the Hebrew University of Jerusalem in 1979. He worked at NYU on the NYU Ultracomputer project in 1980–1982; at the Hebrew University of Jerusalem in 1982–1986; and at the IBM T. J. Watson Research Center from 1986 to 2001. At IBM he headed research that led to the IBM SP scalable parallel system, he contributed to Power 4 and Intel server architecture, and he initiated the Blue Gene project. Dr Snir has published over 100 papers on computational complexity, parallel algorithms, parallel architectures, interconnection networks, compilers and parallel programming environments; he was a major contributor to the design of MPI. Marc Snir is an ACM Fellow and IEEE Fellow. He serves on the editorial board of Parallel Processing Letters and ACM Computing Surveys.

## References


Abandah, G. A. and Davidson, E. S. 1998. Configuration independent analysis for characterizing shared-memory applications. *Proceedings of the 12th International Parallel Processing Symposium*, 485–491, Orlando, Fl.

Aggarwal, A., Chandra, A. K., and Snir, M. 1989. On communication latency in PRAM computations. *Proceedings of the ACM Symposium on Parallel Algorithms and Architectures*, Santa Fe, NM, 11–21.

Aggarwal, A., Chandra, A. K., and Snir, M. 1990. Communication complexity of PRAMs. *Theoretical Computer Science* 71(1):3–28.

Albrecht, A. J. and Gaffney, J. E. Jr. 1983. Software function source lines of code and development effort prediction: a

software science validation. *IEEE Transactions on Software Engineering* 6:639–647.

Allen, F. et al. 2001. Blue Gene: a vision for protein science using a petaflop supercomputer. *IBM Systems Journal* 40(2):310–327.

Anderson, D. P., Cobb, J., Korpela, E., Lebofsky, M., and Werthimer, D. 2002. SETI@home: an experiment in public-resource computing. *Communications of the ACM* 45(11):56–61.

Bailey, D.H. et al. 1995. *The NAS Parallel Benchmarks 2.0,* Report NAS-95-010.

Blelloch, G. E. 1996. Programming parallel algorithms. *Communications of the ACM* 39(3):85–97.

Blelloch, G. E. and Greiner, J. 1996. A provable time and space efficient implementation of NESL. *Proceedings of ACM SIGPLAN International Conference on Functional Programming*, Philadelphia, PA, 213–225.

Blumofe, R. D., Frigo, M., Joerg, C. F., Leiserson, C. E. and Randall, K. H. 1996. An analysis of dag-consistent distributed shared-memory algorithms. *Proceedings of the 8th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA)*, Padua, Italy, 297–308.

Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R. and Selby, R. 1995. Cost models for future software life cycle processes: COCOMO 2.0. *Annals of Software Engineering* 1:57–94.

Brooks, F. P. 1975. *The Mythical Man-Month: Essays on Software Engineering*, Addison-Wesley, Reading, MA.

Culler, D. E., Karp, R. M., Patterson, D. A., Sahay, A., Schauser, K. E., Santos, E., Subramonian, R. and von Eicken, T. 1993. LogP: towards a realistic model of parallel computation. *Proceedings of Principles and Practice of Parallel Programming*, San Diego, CA, 1–12.

Fishburn, P. C. 1970. *Utility Theory for Decision Making*, Lively, New York.

Hey, T. and Lancaster, D. 2000. The development of Parkbench and performance prediction. *International Journal of High Performance Computing Applications* 14(3):205–215.

Hristea, C., Lenoski, D., and Keen, J. 1997. Measuring memory hierarchy performance of cache-coherent multiprocessors using micro benchmarks. *Proceedings of Conference on Supercomputing*, San Jose, CA, 1–12.

JaJa, J. 1992. *An Introduction to Parallel Algorithms*, Addison-Wesley, New York.

Kerbyson, D. J., Alme, H. J., Hoisie, A., Petrini, F., Wasserman, H. J. and Gittings, M. 2001. Predictive performance and scalability modeling of a large-scale application. *Proceedings of Supercomputing Conference (SC01)*, Denver, CO.

McCabe, T. J. and Watson, A. H. 1994. Software complexity. *Crosstalk, Journal of Defense Software Engineering* 7(12):5–9.

McCalpin, J. D. 2004. STREAM: sustainable memory bandwidth in high performance computers, http://www.cs.virginia.edu/stream/.

Peterson, V. L., Kim, J., Holst, T. L., Deiwert, G. S., Cooper, D. M., Watson, A. B. and Bailey, F. R. 1989. Supercomputer requirements for selected disciplines important to aerospace. *Proceedings of the IEEE* 77(7):1038–1055.

Reif, J. H. 1993. *Synthesis of Parallel Algorithms*, Morgan Kaufmann, San Francisco, CA.

RTI. 2002. *The Economic Impacts of Inadequate Infrastructure for Software Testing,* Report 02-3.

Shirts, M. and Pande, V. S. 2000. Screen savers of the world unite! *Science* 1903–1904.

Snavely, A., Carrington, L., Wolter, N., Labarta, J., Badia, R. and Purkayastha, A. 2002. A framework for performance modeling and prediction. *Proceedings of Supercomputing (SC02)*, Baltimore, MD.

Tahvildari, L. and Singh, A. 2000a. Categorization of object-oriented software metrics. *Proceedings of the IEEE Canadian Conference on Electrical and Computer Engineering*, Halifax, Canada, 235–239.

Tahvildari, L. and Singh, A. 2000b. Impact of using pattern-based systems on the qualities of parallel application. *Proceedings of the IEEE International Conference on Parallel and Distributed Processing Techniques and Applications (PDPTA)*, Las Vegas, NV, 1713–1719.

Verner, J. M., Tate, G., Jackson, B., and Hayward, R. G. 1989. Technology dependence in function point analysis: a case study and critical review. *Proceedings of the 11th International Conference on Software Engineering*, Pittsburgh, PA, 375–382.

von Neumann, J. and Morgenstern, O. 1947. *Theory of Games and Economic Behavior,* 1953 edition. Princeton University Press, Princeton, NJ.

Young, H. P. 1994. Cost allocation. *Handbook of Game Theory with Economic Applications*, R. J. Aumann and S. Hart, editors, Elsevier, Burlington, MA, 1193–2235.