

HPC Productivity Model Synthesis

Dr. Jeremy Kepner[†]
kepner@ll.mit.edu
MIT Lincoln Laboratory
244 Wood St., Lexington, MA 02138

March, 2004

Abstract

The DARPA High Productivity Computing System (HPCS) program is developing systems that deliver increased value to users at a rate commensurate with the rate of improvement in the underlying technologies. For example, if the relevant technology was silicon, the goal of such a system would be to double in productivity (or value) every 18 months, following Moore's Law. The key question is how do we define and measure productivity, and what are the underlying technologies that affect productivity. The goal of this paper is to synthesize from several different productivity models a single model that captures the main features of all the models. In addition we will start the process of putting the model on an empirical foundation by incorporating selected results from the software engineering and HPC communities. An asymptotic analysis of the model is conducted to check that it makes sense in certain special cases. The model is extrapolated to an HPC context and several examples are explored, including HPC centers, HPC users, and interactive grid computing. Finally, the model hints at a profoundly different way of viewing HPC systems, where the user must be included into the equation, and innovative hardware is a key aspect to lowering the very high costs of HPC software.

1 Introduction

How do we define productivity in the High Performance Computing (HPC) domain? Traditional measures such as peak floating point operations per second (flops) often do not capture many important factors that affect productivity. This paper seeks to address this question by synthesizing from several different HPC productivity models a single model that captures the main features of all the models and is suitable for empirical analysis.

This paper draws heavily from the several important articles: Snir & Bader 2004; Sterling 2004; Kennedy, Koelbel and Schreiber 2004. The synthesis starts with the model developed by Marc Snir and David Bader based on the economic concept of time dependent utility. Next, some of the more specific definitions of value and cost

[†]This work is sponsored by the Defense Advanced Research Projects Agency under Air Force Contract F19628-00-C-0002. Opinions, interpretations, conclusions, and recommendations are those of the author and are not necessarily endorsed by the United States Government.

developed by Thomas Sterling are included. Finally, the model incorporates the ideas of Ken Kennedy, Charles Koelbel and Rob Schreiber for characterizing productivity of programming environments in terms of their development expressiveness and execution efficiency.

The goals of the model are to provide a unified set of definitions for various concepts found in many of the models and to provide a starting point for further analysis (e.g. looking at special asymptotic cases or by testing with values drawn from empirical studies). The basic notation in the model is that of Snir and Bader. The notation is simplified by only considering the situation of a single representative user programming a representative application on a representative system. It is assumed that all necessary ensemble averaging over users, applications and systems can be performed without loss of generality. Furthermore, this article attempts to provide no additional theoretical motivation for the models. Several new concepts will be introduced; some of the key concepts are:

Productivity is defined as utility over total cost.

Utility which is the value a specific user or organization places on getting a certain answer in a certain time.

Workflow dot product which computes the effort to produce a code by taking the dot product of the workflow vector (fraction of code in each stage) with the programming vector (rate of development at each stage).

Expressiveness quantifies how abstract the programming environment is (e.g., function points/SLOC).

Efficiency which quantifies the computational performance of the programming environment (e.g., % peak).

Factor is an attribute or property of a system or process that directly influences the consequent quality of a system or outcome of a methodology or process.

Parameter is a factor that is quantifiable and may be incorporated in rigorous relations while satisfying constraints of dimensional analysis. A parameter can be binary (e.g. 1 if C++ is supported, 0 otherwise); discrete (e.g., a discrete scale quantifying the quality of C++ support); or continuous. In some cases, a parameter is a stochastic variable rather than a unique value (e.g., the time between system failures).

Metric is a parameter that is observable and measurable. The measurement can be done in an experiment performed on an actual platform (e.g., measuring wall clock time for the execution of a particular program); the measurement can involve humans (e.g. measuring the time needed to code a particular program). Any such measurement has some error that can be estimated. When the measured parameter is a stochastic variable, then one may only estimate some characteristics of the distribution.

These concepts will be integrated into a coherent productivity model throughout the rest of this paper. The basic derivation of the model is presented in section 2. Section 3

gives selected empirical results that can be used in testing the model. Section 4 seeks to test the model by extrapolating the model to a variety of situations including HPC computing centers and HPC programmers. Section 5 provides a conclusion to the paper.

2 Synthesis Model Derivation

In this section we provide a detailed derivation of our newly synthesized productivity model by mapping the results of other formulations onto a common set of definitions.

2.1 Utility Model (Snir)

We begin by defining the productivity Ψ of a system as the time dependent utility, $U(T)$, of the answers it produces divided by the total lifetime cost, C :

$$\Psi \equiv \frac{U}{C} = \frac{U(T)}{C_S + C_O + C_M},$$

where C_S is the software cost, C_O is the cost of ownership, and C_M is the cost of the machine. These costs can also be computed by multiplying the total time, T , by the corresponding rates: $C_S + C_O + C_M = (c_S + c_O + c_M) \times T$. Both the time it takes to get an answer to a particular problem and the costs are strong functions of system parameters P and the application characteristics Q (i.e. $T = T(P, Q)$ and $C = C(P, Q)$).

Utility is the value a user places on getting a result at time T . In general, Utility Theory is applied to situations where one person can place a value on an item, but can't exchange it with another group or person (i.e. there is no efficient market for the item.) The answers people obtain from HPC systems are valuable to them but they can't buy them or sell to others. Thus the units of Utility are "output" as defined by the user, and this could be measured in time, money, flops, scientific papers, or nobel prizes. Figure 1 shows several possible utility curves that may correspond to different types of HPC users. The precise shapes of these curves are unknown and remain to be determined.

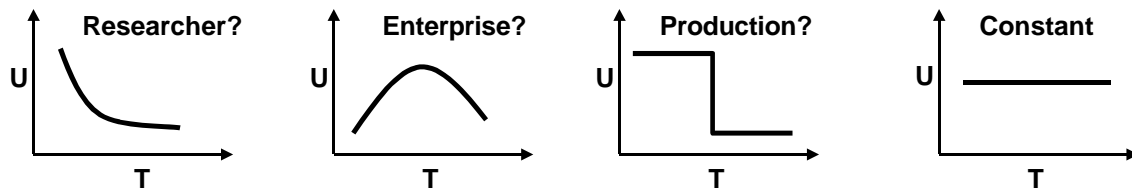


Figure 1: Possible Utility Curves. Utility curves of various possible HPC users.

2.2 Special Theory of Productivity (Sterling)

The above productivity model can be made more specific by defining the utility as useful work (e.g. operations) divided by the time

$$U(T) = W/T = S_p E A$$

where

W = total work done (e.g. total operations)

S_p = peak processing speed (operations/time)

E = efficiency

T_A = time available

T = time

$A = T_A / T$ = availability.

Furthermore, software cost can be roughly approximated by the code development rate times the size of the program times the cost of a programmer

$$C_S = c_f \Gamma r$$

where

c_f = programmer cost (\$/time)

Γ = program size (e.g. SLOC)

r = programming effort (e.g. time/SLOC).

Inserting these terms into the productivity expression yields

$$\Psi \equiv \frac{S_p E A}{(c_f \Gamma r) + C_O + C_M},$$

A further refinement of the model is obtained by computing the programming effort as a weighted sum over the workflow $r = \sum r_i \rho_i$, where r_i is the programming effort at the i th stage (e.g. $i \in$ (design, code, test) \times (new, reuse) \times (serial, parallel)) and ρ_i is the fraction of the total development that goes through that stage.

2.3 Expressiveness and Efficiency (Kennedy, Koelbel & Schreiber)

One of the main goals of the productivity model is to get insight into different technological tradeoffs. In the area of software one of the largest tradeoffs is between the expressiveness or level of abstraction provided by a language, library or tool and the computational execution efficiency it provides. It is often the case that the software environments that are the easiest for the developer provide much lower execution efficiency. This tradeoff can be quantified by looking at the ratios of the execution time and development time achieved using standard practices (unprimed quantities) as compared to those using a new software environment (primed quantities). These ratios define the development expressiveness and execution efficiency intrinsic to the programming environment

$$P_{dev} = T_{dev} / T'_{dev}$$

$$P_{exe} = T_{exe} / T'_{exe}$$

where

P_{dev} = programming tool development power (abstract expressiveness)

P_{exe} = programming tool execution efficiency (computational efficiency)

T_{dev} = development time

T_{exe} = execution time

2.4 (SK)³ Synthesis Model

We are now in a position to combine all of the above idea into a single synthesis formulation. Substituting terms from earlier expressions, we can approximate development time by (Γr) and the ratio of the development reduces to the ratio of the efficiencies

$$P_{dev} = T_{dev}/T'_{dev} = (\Gamma r)/(\Gamma r)'$$

$$P_{exe} = T_{exe}/T'_{exe} = E'/E$$

Substituting these into the productivity formula, we can now estimate the productivity of a new environment as

$$\Psi' \equiv \frac{S_p P_{exe} E A}{C_S/P_{dev} + C_O + C_M} \quad (SK)^3$$

The above formula represents the synthesis of models from many of the individual authors and will be referred to as the (SK)³ formula hereafter.

3 Empirical Foundations

In the previous section a theoretical formulation was presented based on economic definitions of productivity, dimensional analysis of work and cost, and insight about execution time versus development time tradeoffs that occur in HPC software development. Moving the model to the next level requires empirical data that allows us to accept or reject its validity. Unfortunately, there is very little data that is specific to HPC. This section pulls together the few empirical results that exist relating to HPC development time effort. First, a few relevant empirical results will be presented from the mainstream software engineering community. Second, we present several anecdotal results from the HPC community that later will be used to estimate certain model parameters. These results will then be used in the subsequent section as a basis of extrapolation into a variety of HPC programming regimes about which some intuition exists.

3.1 Relevant Software Engineering Results

In this section we select and present several relevant results drawn from the empirical software engineering community. These results can be used to roughly estimate some of the parameters in the productivity model, but they must be used with great care since we are taking results from the mainstream software engineering community and applying them to the HPC community.

The first result (adapted from Boehm 1981) concerns software development activities. Specifically, the amount of effort spent doing Analysis and Design, Coding and Auditing, and Checkout and Test. The data in Table 1 were collected over many years from several NASA projects

<u>Project</u>	<u>Analysis & Design</u>	<u>Coding & Auditing</u>	<u>Checkout & Test</u>
SAGE	39	14	47
NTDS	30	20	50
Gemini	36	17	47
Saturn V	32	24	44
OS/360	33	17	50
TRW Survey	36	20	34

Table 1: Workflows (adapted from Boehm 1981). Percentages of time spent in different stages of software development for several NASA projects.

The principal conclusion of this data is that a large fraction of the software development effort is spent testing. This result is probably a lower bound for HPC software since most HPC testing must be done at multiple scales and even on multiple systems. In addition, the above result has strong implications for software reuse and maintenance. Specifically, even a small code change may require repeating many tests, resulting in a large effort.

The second result (adapted from Boehm et al 1998) relates to the effort it takes to develop a code as a function of the size of the code produced. When estimating the size of a program, there are many possible metrics. Unfortunately, all are imperfect. The most widely used metric is delivered (i.e., fully tested) Source Lines Of Code (SLOCs), which has the advantage of being a well defined and repeatable metric. SLOCs can be thought of as rough estimate of the size of a program and it is expected that total effort would be a monotonically increasing function of size. This function is expressed by the Constructive Cost Model (COCOMO) as

$$\text{Effort} = A (\#\text{SLOC})^B$$

The coefficients A and B are functions of the difficulty of the project, experience of the programmers, and a variety of other factors. A rule of thumb for large, complex government systems such as air traffic and military control systems is 8 SLOCs per day (i.e., $A = 1 \text{ day}/8 \text{ SLOCs}$, $B = 1$). For standard commercial code written in C++ the rule of thumb is closer to 20 SLOCs per day. For student Java code with very few requirements the rate might be closer 70 SLOCs per day. Typically these estimates are within 50% of the true value. SLOC metrics must be used with great care; SLOCs/day are averages generated from many people over many projects and do not reflect all people, or all projects. These averages are good for roughly estimating the cost of a project and for potentially evaluating technologies, but SLOCs should not be used to evaluate programmers as some of the best programmers are the ones who can do a job using fewer lines of code but may take longer to write them.

The simple linear cost model indicates that one of the important ways to shorten the length of a project is to reduce the number of lines of code. One of the most common ways to do this is through code reuse. Code reuse leverages the design and coding stages for the reused lines; however, it may not eliminate testing. In fact, the utility of reused code is a function of how much of the reused code needs to be rewritten and retested. If all the code needs to be rewritten, then there is no benefit to code reuse. If none of the code needs to be rewritten then the benefit of the reused code is 100%. This third result from software engineering is shown in Figure 2 (adapted from Selby 1988), and indicates

that the cost of reusing goes up rapidly as the fraction of the code that needs to be rewritten increases.

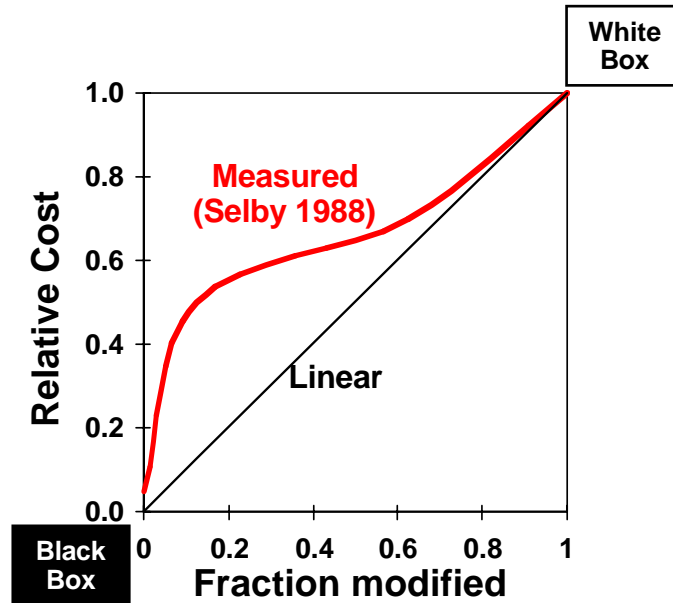


Figure 2: Cost of Reuse. The relative cost (compared to writing from scratch) of reused code as a function of the fraction of the code that must be modified. Code that can be treated as a “black box” can be reused at almost no cost. A small fraction of modifications (20%) can drive the cost up dramatically.

The implications of this result for HPC programming are quite significant. If HPC code can be treated as a “black box” then the benefits of reuse can be obtained. Unfortunately, in order to achieve performance, reused HPC code often needs to be integrated and even restructured at the lowest level to minimize data movement, which results in “white box” code that is costly to reuse. Therefore, for HPC to get the full benefits of reuse, it must develop technologies (e.g., the Common Component Architecture) which allow pieces to be reused while maintaining performance.

The fourth result from software engineering that we discuss deals with the issue of reducing SLOCs from a different perspective. Specifically, the abstraction level of the language. Programming languages differ dramatically in the level of detail they require from the programmer. Low level languages such as assembly or VHDL require the programmer manipulate data at the register or even gate level of the processor. High level languages such as Java or Matlab allow the user to accomplish more functionality with fewer lines of code by giving up control over the precise details of the calculation. In order to measure the abstractness of a particular language, we need an independent measure of functionality. The most commonly used such measure is the function point, which counts the number of inputs and outputs that exist within a program in a weighted fashion. Table 2 (adapted from Jones 1991) gives SLOCs per function point for several languages.

<u>Language</u>	<u>Approximate SLOCs per Function Point</u>
C	100
Fortran	100
C++	30
Java	30
Matlab	10
Python	10
Spreadsheet	5

Table 2: SLOCs per Function Point (adapted from Jones 1991). Rough estimates of the number of lines of code per function point for several languages. For any specific code, these numbers can vary widely due to the presence or absence of specific features in a language that may be particularly well suited to a given problem.

The implications of these results for HPC programming are also significant. If HPC code can be rewritten using a high level language then the benefits of abstraction can be obtained. Unfortunately, in order to achieve performance, HPC code often needs to be written at a lower level which can limit the effectiveness of high level languages. In addition, it must be noted that estimates in Table 2 are rough averages. For a specific program, the number of lines of code can be strongly affected by the specific features in a language and how well suited it is to a given problem (e.g. Fortran array constructs, C bit level operations, Matlab linear algebra operations).

3.2 Anecdotal HPC results

The previous subsection gave a brief summary of several classical results taken from the empirical software engineering community. It should be emphasized that these results were not drawn from HPC codes and therefore great care should be taken when generalizing from mainstream software development to HPC software development. In this section we present an overview of selected anecdotal HPC results, which in some cases support and in some cases contradict the results in the previous sections.

The first anecdotal result is drawn from the Department of Energy's Accelerated Strategic Computing Initiative which is currently the largest HPC effort in the world. ASCI has many large codes that are run on some of the World's largest computers. Table 3 (adapted from Post & Kendall 2004) gives approximate size, effort, and SLOC rate data on six codes taken from this program at different stages of maturity and completion. These data should be viewed as incomplete. Never-the-less, in spite of the differences, the SLOC rates are typical of what is seen in the broader community. Most of the codes are being written at rates that are typical of the broader software community and some codes differ dramatically. Usually large differences are due to differences in schedule and operating requirements. For example a high code rate may be a reflection of a significant amount of code reuse and a low code rate may be a reflection of a code that is being run because it is producing useful answers.

<u>Code</u>	<u>SLOCs</u>	<u>Team Size</u>	<u>Age (years)</u>	<u>SLOCs/man-day</u>
ASCI A	184,000	20	3	12
ASCI B	640,000	22	9	13
Antero	300,000	17	4	18
Shevano	500,000	8	3.5	71
Crestone	314,000	12	8	13
Blanca	200,000	35	8	3

Table 3: ASCI Codes (Adapted from Post and Kendall 2004). Approximate size and effort data for several large HPC codes. These results are typical of what is found in the broader community. Many of the codes are being developed at typical rates while other differ dramatically (usually due to different schedule and operating requirements).

The second anecdotal result is drawn from the NAS Parallel Benchmark suite (Bailey et al 1994). The NAS benchmarks consist of five computational kernels and three small applications that were designed to test HPC performance. In addition, they are unique in that they were developed from a precise written specification which has allowed them to be implemented in a variety of languages. These benchmarks are a unique opportunity to examine the coding effort required from different parallel programming environments to write small ~1000 SLOC codes. It is not clear how to extrapolate these results to larger codes. Table 4 and Figure 3 gives the relative size of the eight codes and the ratio of the serial code (written in C or Fortran) to parallel versions implemented in MPI, OpenMP, HPF, and parallel Java.

Implementation	BT	CG	EP	FT	IS	LU	MG	SP	Avg
Serial SLOCs	2,528	574	141	560	450	2,554	863	2,120	-
MPI/Serial	1.67	1.83	1.54	2.54	1.63	1.38	1.92	1.54	1.77
OpenMP/Serial	1.00	1.02	1.25	1.34	1.17	1.03	1.00	1.04	1.10
HPF/Serial	1.22	1.06		1.13		1.14	1.21	1.06	1.13
Java/Serial	1.62	1.16		1.61	0.85	1.78	1.72	1.80	1.50

Table 3: NAS Parallel Benchmarks (Funk Personal Communication). Size of serial code and relative sizes of corresponding parallel implementations. Consistently the MPI code is the largest, followed by Java and HPF and OpenMP.

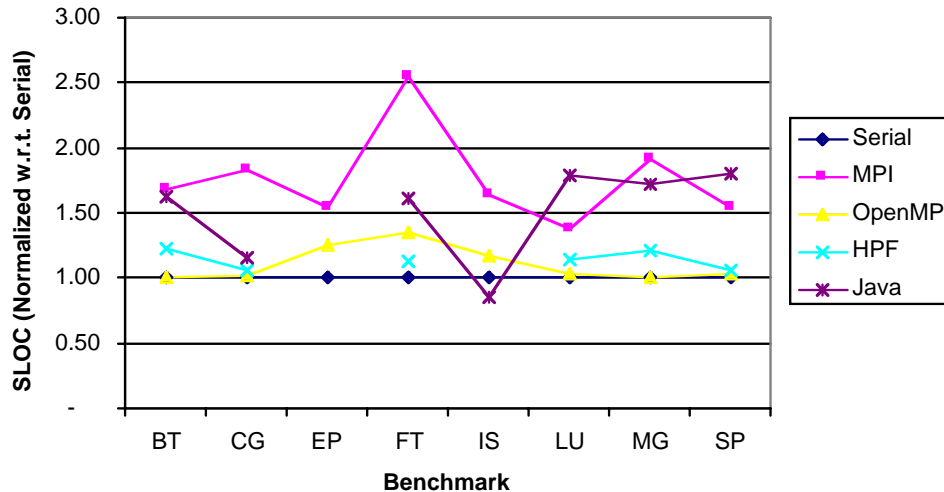


Figure 4: NAS Parallel Benchmarks (Funk personal communication). Graphical depiction of the relative sizes of different parallel codes relative to serial codes.

The NAS benchmark data presents a remarkably consistent picture of the coding overhead associated with these different parallel programming environments. The data indicate that MPI consistently incurs a large code overhead (70%), while OpenMP and HPF incur relative small code overheads (10%). Drawing conclusions from these ratios can be very misleading. For example, often programmers are able to encapsulate the communication related parts of a code, and isolate them from the computations. The net result is that the MPI overhead may be 70%, but only in the 10% of the code that deals with boundary conditions.

Another interesting result is the relatively high overhead of Java. Based on the function point data in Table 2, we would have expected a serial version in Java to be 1/3 the size of the C or Fortran code. The fact that Java is much larger is an example of the specific features of the languages being mismatched to the problem. Specifically, the NAS benchmarks are mainly structured around regular multi-dimensional arrays of a type that Java does not readily support.

The issue of how well suited a language is to a problem is more fully illustrated in Chamberlain et al 2000, in which they implement the NAS MG benchmark in parallel ZPL. ZPL is a high level language with extensive support for mathematics on multi-dimensional arrays. The SLOCs of the ZPL NAS MG implementation is 20% the size of the serial code and represents a significant reduction. The relative sizes of the different implementations of NAS MG are shown in Figure 6 (Chamberlain personal communication), along with a breakdown of code between communications, declarations and computations. The data suggest that the large overhead of MPI is primarily due to explicit communications directives (as expected) and the large Java overhead is due to increased number lines required to implement the computation.

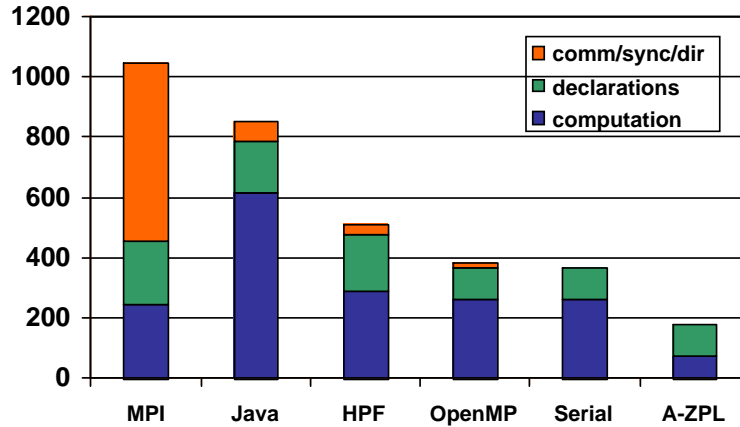


Figure 6: NAS MG Benchmarks (Chamberlain personal communication). Code breakdown.

The NAS benchmarks are notable in their ability to provide insight into the coding overhead of different parallel programming environments. Of course, their primary aim is to look at execution performance. Comparison of different benchmark implementations is difficult because of the necessary expertise required to obtain good performance in a variety of different environments. One intriguing example is shown in Figure 7, in which the NAS MG ZPL implementation is compared with the Fortran77 MPI implementation.

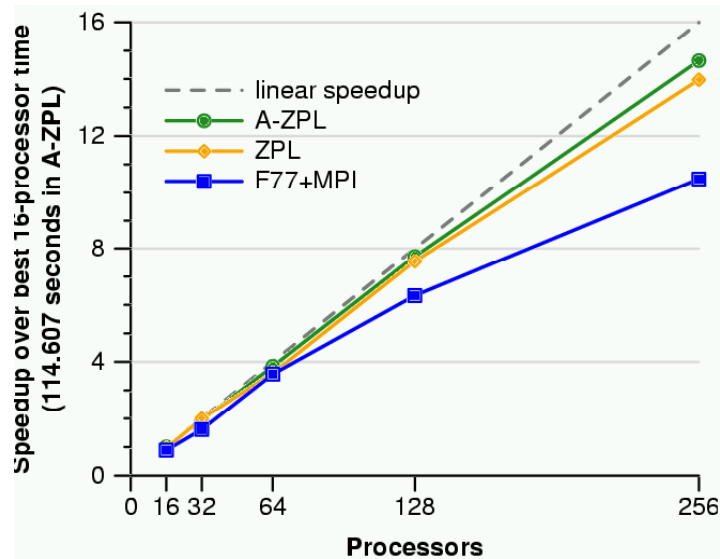


Figure 7: NAS Parallel Benchmarks (Chamberlain personal communication). Comparison of ZPL and MPI implementations on different numbers of processors.

Ultimately, we would like to be able obtain data on both the coding overhead and execution performance that provide quantitative insight into the potential tradeoffs that can be made between the two. Figure 8 is an example of a single parallel image processing kernel implemented by the same programmer using seven different programming environments. The data clearly suggest that there is a tradeoff space

between different environments for this particular application. However, it remains to be seen if these kind of results can be generalized beyond a specific programmer and a specific application.

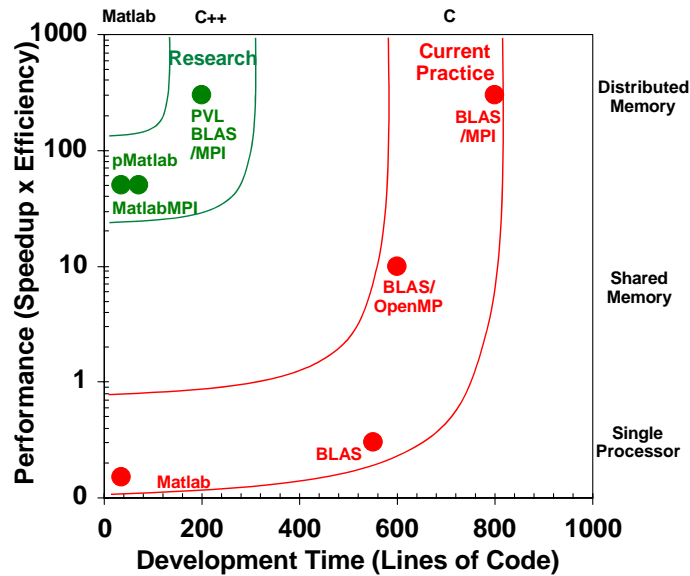


Figure 8: Performance vs. Effort. Data for an image filtering kernel implemented by the same programmer using four standard environments and three research environments.

Both the ZPL and pMATLAB experiments involve relatively small amounts of code (hundreds of lines), and it is hard to draw conclusions that would be relevant to codes with hundreds of thousands of lines. Never-the-less, these results are interesting and suggest that better approaches may exist.

4 Empirical Analysis and Extrapolation

The previous section has attempted to provide some rough estimates of some of the parameters found in the $(SK)^3$ formulation. In this section we attempt several simple tests of the formula to see if it remains plausible. Currently, there is insufficient empirical data from the HPC community to conduct rigorous tests of these models. Thus, we have to extrapolate based on a limited amount of data, which provides severe limits on the validity of any results, and they should not be generalized at this point until a firmer empirical footing is established. First, an asymptotic analysis is done in which several highly simplified versions of the formula are applied to “real” situations and the resulting formulations are checked for their validity. Second, using the best current estimates of the effort required to use messaging and threaded programming models, we apply the formulas to an HPC context and check the validity of these results. Finally, we look at how to apply the model to interactive grid computing.

4.1 Asymptotic Analysis

In this section four special case “sanity checks” are conducted on the $(SK)^3$ formulation. These four cases correspond to simplified versions of the following hypothetical situations:

- Supercomputing center trying to get into the Top500
- Office worker who wants to run Microsoft Office
- Software developer who wants to reduce programming time
- Estimating the benefit of a programming tool

Each of these situations can be represented by fairly simple modifications to the full formula.

In the supercomputing center scenario we make the following assumptions. First, the utility is simply the total operational rate on the benchmark

$$U = S_p E A$$

We set $E = 1$ and $A = 1$ since the Top500 benchmark can usually achieve near 100% efficiency on many architectures and only needs to be run once. Thus, the utility is simply the theoretical peak of the system: $U = S_p$. Software cost is unimportant in this instance and can be dropped ($C_S = 0$). Operating costs for a supercomputing center are usually a fixed cost. Finally, the amount of money available for purchasing the system is also usually fixed. In this case, the productivity formula reduces to

$$\Psi_{\text{Top500}} = S_p / (C_M = \text{fixed})$$

According to this function, to maximize its productivity goal (i.e. Top500) a supercomputing center will seek the highest theoretical peak system for given machine budget.

In the office worker scenario the utility of the computer is determined by its ability to run MS Office

$$U = \begin{cases} 1: & \text{if(MS Office)} \\ 0: & \text{Otherwise} \end{cases}$$

The software cost is that of the software that comes bundled with the hardware. The maintenance costs for desktop system are typically 5x the hardware, but are typically ignored at the time of purchase, which yields

$$\Psi_{\text{OfficeWorker}} = 1 / C_M$$

According to this function, to maximize its productivity goal (i.e. run MS Office) an office worker will seek the lowest cost machine that can run the software.

In the software developer scenario, the programmer is principally concerned with the time it takes them to write programs. The basic utility of a machine is reflected by its ability to run their development environment, for example C++

$$U = \begin{cases} 1: & \text{if(C ++ environment)} \\ 0: & \text{Otherwise} \end{cases}$$

The typical cost of the software they write can be represented by

$$C_S = (\$200\text{K/year}) \Gamma r$$

Assume there is one system administrator per 20 programmers, in which case the operating costs are

$$C_O = (\$200\text{K}/\text{year}) / 20 = \$10\text{K}/\text{year}$$

Furthermore, assume the programmer buys one desktop computer every three years

$$C_M = (\$3\text{K})/(3 \text{ year}) = \$1\text{K}/\text{year}$$

Combining terms and noting that $C_S \gg C_O + C_M$, we can approximate the programmer productivity function by

$$\Psi_{\text{Programmer}} = 1 / C_S$$

According to this function, to maximize their productivity goal (i.e. write software faster) the programmer will seek systems that reduce the cost of writing programs.

The final scenario builds on the previous one by looking at the value of a programming environment to a programmer. Let Ψ be the productivity of the programmer without the programming tool, and let Ψ' be the productivity of the programmer using a tool with development power P_{dev} . Also, assume $\Psi' > \Psi$. Based on the previous scenario, we have

$$C_S/P_{\text{dev}} + C_O + C_M + C_{\text{tool}} < C_S + C_O + C_M$$

Canceling like terms yields

$$C_S/P_{\text{dev}} + C_{\text{tool}} < C_S$$

Or,

$$P_{\text{dev}} > C_S/(C_S + C_{\text{tool}})$$

Many professional developer environments sell for ~\$40K. If we assume that this investment is recovered in one year by increased productivity, then this yields the following bounded estimate

$$P_{\text{dev}} > 1.2$$

which implies that a \$40K tool should increase programmer productivity by 20% in order to justify its cost.

In this section we have examined the productivity formulation in four simple user cases with simple goals. In each case the formulation reduces to a formula that captures the essence of the goals and is in agreement with intuition.

4.2 Extrapolation to HPC

Currently, there is insufficient empirical data from the HPC community to conduct rigorous tests of these models in an HPC context. In this section we attempt to use the best current estimates of the effort required to use messaging and threaded programming models in order to apply the formulas to HPC settings and further check the validity of the formulation.

To begin the analysis, consider the productivity of a programmer writing and running a serial program on a workstation. Assume that their utility is measured in usable flops and that they achieve 25% efficiency. Also assume the workstation costs \$2K, has a peak

of 4 GFlops, is available 99% of the time, and costs \$10K a year to operate. Finally, assume the programmer costs \$200K per year. These values correspond to the following model parameters:

$$S_p \sim 4 \text{ GFlops}, E \sim 25\%, A \sim 0.99, c_f \sim \$200\text{K/yr}, C_O = \$10\text{K}; C_M = \$2\text{K}$$

Now consider the case of the same programmer porting their code using MPI to run on a commodity cluster. For simplicity, assume that the application scales perfectly using MPI and that the availability of the parallel system is the same as the workstation. Furthermore, assume the hardware costs scale linearly with the number of processors and that operating costs scale with the size of the cluster but at a much smaller slope (e.g. 0.1). Finally, assume the MPI code is 1.7 times the size of the serial code, and that these lines of code are written at half the rate of the serial code (primarily due to the additional testing required). These values correspond to the following model parameters

$$P_{\text{eff}}(\text{MPI}) \sim N, A^{\parallel} \sim A, P_{\text{abs}}(\text{MPI}) \sim 1/3, C_O^{\parallel} = 0.1 N C_O, C_M^{\parallel} = N C_M$$

Inserting these expressions into the productivity formula yields (in terms of the serial values):

$$\Psi_{\text{MPI}} \approx \frac{N S_p E A}{3C_s + 0.1 N C_O + N C_M}$$

The next case we consider is the same programmer porting their code using OpenMP on a large shared memory system. Again, assume that the application scales perfectly using OpenMP and that the availability of the parallel system is the same as the workstation. Let the hardware costs scale linearly with the number of processors, but each node is 3x more expensive. Because it is a single system, assume the operating costs are fixed at 5 times the cost of a workstation. Finally, assume the OpenMP code is 1.1 times the size of the serial code, and that these lines of code are also written at half the rate of the serial). These values correspond to the following model parameters

$$P_{\text{eff}}(\text{OpenMP}) \sim N, A^{\parallel} \sim A, P_{\text{abs}}(\text{OpenMP}) \sim 1/1.2, C_O^{\parallel} = 5 C_O; C_M^{\parallel} = 3 N C_M$$

Inserting these expressions into the productivity formula yields

$$\Psi_{\text{OpenMP}} \approx \frac{N S_p E A}{1.2C_s + 5C_O + 3N C_M}$$

In order to obtain quantitative values from these formulas, we need to make two additional assumptions: the size of the system and the size of the code. We shall do this by considering two specific cases, in each case we consider the base case scenario when the program scales perfectly and the worst case scenario of when the program fails to scale at all. In the first case consider the productivity of a 1000 processor system with respect to a large 100,000 line program, which yields:

$$\Psi_{\text{OpenMP}(\text{best})} = 1 \text{ TFlops} / [\$3.6\text{M} + \$50\text{K} + \$6\text{M}] = 100 \text{ KFlops}/\$$$

$$\Psi_{\text{MPI}(\text{best})} = 1 \text{ TFlops} / [\$10\text{M} + \$1\text{M} + \$2\text{M}] = 80 \text{ KFlops}/\$$$

$$\Psi_{\text{serial}} = 1 \text{ GFlops} / [\$3\text{M} + \$10\text{K} + \$2\text{K}] = 0.3 \text{ KFlops}/\$$$

$$\Psi_{\text{OpenMP}(\text{worst})} = 1 \text{ GFlops} / [\$3.6\text{M} + \$50\text{K} + \$6\text{M}] = 0.1 \text{ KFlops}/\$$$

$$\Psi_{\text{MPI}(\text{worst})} = 1 \text{ GFlops} / [\$10\text{M} + \$1\text{M} + \$2\text{M}] = 0.08 \text{ KFlops}/\$$$

In the second case consider the productivity with respect to a small 1,000 line code on a 1,000 processor system which the user has free access to (e.g. at a center):

$$\Psi_{\text{OpenMP}(\text{best})} = 1 \text{ TFlops} / [\$36\text{K}] = 27,000 \text{ KFlops}/\$$$

$$\Psi_{\text{MPI}(\text{best})} = 1 \text{ TFlops} / [\$100\text{K}] = 10,000 \text{ KFlops}/\$$$

$$\Psi_{\text{serial}} = 1 \text{ GFlops} / [\$30\text{K}] = 33 \text{ KFlops}/\$$$

$$\Psi_{\text{OpenMP}(\text{worst})} = 1 \text{ GFlops} / [\$36\text{K}] = 27 \text{ KFlops}/\$$$

$$\Psi_{\text{MPI}(\text{worst})} = 1 \text{ GFlops} / [\$100\text{K}] = 10 \text{ KFlops}/\$$$

These results were obtained by making a lot of unproven assumptions and so great care should be made in their interpretation and they should not be generalized at this time. Nevertheless, the results lend themselves to the following observations. First, the formulas do seem to allow us to generate reasonable numbers using relatively simple assumptions. Second, the values are all in terms flops/\$ (a familiar unit to the HPC community), but are much lower than we are typically used to seeing because the large cost of the software is included. Third, if performance is achieved (best), then a significant productivity benefit is realized, however, if it is not achieved then the productivity is lower than serial case (worst).

At a higher level, this model is hinting at something profound in the way we view HPC systems. Specifically, we see that there is an enormous range of productivity values, which reinforces the notion that value a user obtains from a system is specific to them, and that measuring the value of system independent of the user is problematic. Finally, and perhaps even more important, this model shows a way to include the significant impact of software cost into evaluating a system. These costs are often ignored, but may be the primary driver justifying innovative hardware. In other words the reason we invest a lot of money in hardware is to help lower the even higher cost of the software.

4.3 Extrapolation to Interactive Grid Computing

In this section, the model is applied to the emerging area of interactive grid computing. Although a new area, in many respects it is easier to calculate the productivity of such systems. HPC computing is often focused on “capability” computing whereby fundamentally different calculations can be done when a program is run on thousands of processors. Grid computing is more focused on “capacity” computing, which simply allows current programs to run faster, often in an embarrassingly parallel manner. Interactive grid computing provides access to this capability in an on-demand and interactive fashion. In such a situation, the utility can be expressed as simply the value of the time saved by the user. For a single user running a single job the time saved is

$$T - T/S = T(1 - S^{-1}) \sim T(1 - N_{\text{CPU}}^{-1}) ,$$

where S is the parallel speedup and N_{CPU} is the number of processors. For the whole system, utility is the value of programmer time multiplied by the time saved by all users

$$U(T) = c_f \int^T dt \sum_{\text{user} \in \text{Users}(t)} 1 - S_{\text{user}}^{-1},$$

where $\text{Users}(t)$ are the set of user running at time t . This formula can be approximated with the average number of users, N_{AvgUsers} , and the average number of processors, N_{AvgCPU} , used over time T

$$U(T) \sim c_f T N_{\text{AvgUsers}} (1 - N_{\text{AvgCPU}}^{-1}).$$

Interestingly, this formula suggests that for interactive grid computing, it is better to have more users running smaller jobs than one user running on the whole system.

The costs can be divided up into user costs and grid costs. Users costs include training time, $c_f N_{\text{TotUsers}} T_{\text{train}}$, the time it takes to launch on the grid, $c_f N_{\text{launch}} T_{\text{launch}}$, and the time to make the software parallel so it can run on the grid

$$c_f N_{\text{TotUsers}} \Gamma r (1/P_{\text{dev}} - 1).$$

The grid costs include the time of the operators $c_f T_O$, and the cost of the hardware $c_f T_M$, which for convenience is expressed in terms of equivalent personnel costs. Adding these terms gives the total costs

$$c_f \{ N_{\text{TotUsers}} [\Gamma r (1/P_{\text{dev}} - 1) + T_{\text{train}}] + N_{\text{launch}} T_{\text{launch}} + T_O + T_M \}.$$

Even without going into the details, it is clear that certain terms have the potential for becoming very large. For example, training costs are multiplied by *all* users, not just those using the system. Parallel coding effort can also be quite large since it can potentially require all users to rewrite a significant fraction of their code (e.g. if $P_{\text{dev}} \sim 1.5$). Finally, launch time is also important since it is multiplied by every launch, which could be in the tens of thousands of launches.

Dividing the utility by the costs, gives the full interactive grid productivity model

$$\Psi_{\text{RTgrid}} \approx \frac{c_f \int^T dt \sum_{\text{user} \in \text{Users}(t)} 1 - S_{\text{user}}^{-1}}{c_f \{ N_{\text{TotUsers}} [\Gamma r (P_{\text{dev}}^{-1} - 1) + T_{\text{train}}] + N_{\text{launch}} T_{\text{launch}} + T_O + T_M \}},$$

which can be approximated by

$$\Psi_{\text{RTgrid}} \approx \frac{T N_{\text{AvgUsers}} (1 - N_{\text{AvgCPU}}^{-1})}{N_{\text{TotUsers}} [\Gamma r (P_{\text{dev}}^{-1} - 1) + T_{\text{train}}] + N_{\text{launch}} T_{\text{launch}} + T_O + T_M}.$$

Now consider a small corporate grid facility that acquires about 200 CPUs/year, averages 20 users each running on 10 CPUs, and is maintained by one full time employee. Furthermore, assume the programs are easy to make parallel ($P_{\text{dev}} = 1/1.01$), training time is small (4 hours), and launch time is near zero. The parameters corresponding to this situation are

- $T = 2000$ hours
- $N_{\text{AvgUsers}} \sim 20$
- $N_{\text{AvgCPU}} \sim 10$
- $N_{\text{TotUser}} \sim 100$
- $\Gamma \sim 1000$ lines
- $r \sim 1/(2 \text{ lines/hour})$

$$P_{\text{dev}} = 1/1.01$$

$$T_{\text{train}} \sim 4 \text{ hours}$$

$$N_{\text{launch}} \sim 10,000$$

$$T_{\text{launch}} \sim 0$$

$$T_{\text{O}} \sim 2000 \text{ hours}$$

$$T_{\text{M}}(200 \text{ CPUs}) \sim 4000 \text{ hours}$$

Inserting these numbers into the interactive grid productivity formula yields a value of 5.2, which in this case means that for every \$1 spent on the grid, \$5.2 are returned in user time saved.

In this section, the $(SK)^3$ model has been applied to interactive grid computing, and the model is used to compute return on investment for a typical grid installation. The results are consistent with intuition. In addition, the model highlights the need for controlling users costs associated with training, porting and launching as these all have very large multipliers.

5 Conclusions

The DARPA High Productivity Computing System (HPCS) program is developing systems that deliver increased value to users at a rate commensurate with the rate of improvement in the underlying technologies. The key questions are how do we define and measure productivity, and what are the underlying technologies that affect productivity. This paper has synthesized from several different productivity models a single model that captures the main features of all the models. In addition the process of putting the model on an empirical foundation has been started by incorporating selected results from the software engineering and HPC communities. An asymptotic analysis of the model was conducted to check that it makes sense in certain special cases. The model was extrapolated to several HPC examples and the results were found to be consistent with our intuition about HPC systems. Finally, this model is hinting at a profoundly different way of viewing HPC systems

- The value provided by an HPC system is highly user specific and the user must be included in the equation
- The reason we invest a lot of money in hardware is to help lower the even higher cost of the software

6 Acknowledgements

I would like to thank Bob Graybill (DARPA) and Fred Johnson (DoE Office of Science) for their stimulation of the area of HPC Productivity. I would also like to thank the authors of the other models upon whom much of this work was based: David Bader, Thomas Sterling, Marc Snir, Rob Schreiber, Charles Koelbel and Ken Kennedy. I would also like to thank Vic Basili, Brad Chamberlain, Doug Post and Andy Funk for their valuable contributions. I would like to thank several anonymous referees for their significant contributions. Finally I would like to thank Bob Bond whose vision in this area was the inspiration for this work.

7 References

D Bailey, E Barszcz, J Barton, D Browning, R Carter, L Dagum, R Fatoohi, S Fineberg, P Frederickson, T Lasinski, R Schreiber, H Simon, V Venkatakrisnan, and S Weeratunga, The NAS Parallel Benchmarks, RNR Technical Report RNR March 1994

Boehm, B., Software Engineering Economics, Prentice Hall, 1981

Boehm, B., COCOMO II Model Definition Manual, USC, 1998

Chamberlain B., Deitz, S., Snyder, L., A Comparative Study of the NAS MG Benchmark across Parallel Languages and Architectures, In Proceedings of the 2000 ACM/IEEE Supercomputing Conference on High Performance Networking and Computing (SC2000), November 2000

Chamberlain, B., 2003, Personal Communication

Funk, A. 2003, personal communication

Jones, Capers, Applied Software Measurement, McGraw-Hill, 1991, ISBN 0-07-032813-7

Kennedy, K., Koelbel, C. and Schreiber, R., International Journal of High Performance Computing and Applications: Special Issue on HPC Productivity (editor: Kepner), Volume 18, Number 4, Winter 2004 (November)

Post, M. and Kendall, R., International Journal of High Performance Computing and Applications: Special Issue on HPC Productivity (editor: Kepner), Volume 18, Number 4, Winter 2004 (November)

Selby, R.. Empirically Analyzing Software Reuse in a Production Environment. In Will Tracz, editor, Software Reuse: Emerging Technology, IEEE Computer Society Press, 1988, pages 176-189.

Snir, M. and Bader, D., International Journal of High Performance Computing and Applications: Special Issue on HPC Productivity (editor: Kepner), Volume 18, Number 4, Winter 2004 (November)

Sterling, T., International Journal of High Performance Computing and Applications: Special Issue on HPC Productivity (editor: Kepner), Volume 18, Number 4, Winter 2004 (November)