Measuring Productivity on High Performance Computers

Marvin Zelkowitz[1,2] Victor Basili[1,2] Sima Asgari[1]
Lorin Hochstein[1] Jeff Hollingsworth[1] Taiga Nakamura[1]
[1]Department of Computer Science, University of Maryland, College Park, MD 20742
[2]Fraunhofer Center Maryland, College Park, MD 20740
{mvz, basili, sima, lorin, hollings, nakamura}@cs.umd.edu

Abstract

In the high performance computing domain, the speed of execution of a program has typically been the primary performance metric. But productivity is also of concern to high performance computing developers. In this paper we will discuss the problems of defining and measuring productivity for these machines and we develop a model of productivity that includes both a performance component and a component that measures the development time of the program. We ran several experiments using students in high performance courses at several universities, and we report on those results with respect to our model of productivity.

1. Introduction

Productivity is an economic concept that measures the amount of output per unit of input in an economic system. For a unit of output we will use the general concept of *value* and for a unit of input we will use the concept of *cost*. Therefore, a simple definition of productivity is *value/cost*. In this paper we want to define these terms in the context of solving specific problems on a computer, in particular, for the domain of high performance computers.

This seemingly simple relationship of *value* divided by *cost* becomes quite complex when we try to quantitatively define *value*. The value of a product depends upon the needs of the various stakeholders involved in creating or using a product. As we show, under different situations, the value of a product may change depending upon context.

In this paper, we present the development of a productivity measure applicable for the high performance computing (HPC) domain. Since we want to compute productivity values for specific programs, our goal is not only a formal model of productivity, but one that has parameters that can be easily computed. While our goal is to ultimately produce an absolute measure of productivity, in this paper we limit our discussion to the relative productivity between two implementations. Even this limited form of productivity should be useful, such as deciding which computer system or development method would be best for solving a specific application problem.

Since the value component of productivity depends upon the needs of the various stakeholders, we can approach productivity from several points of view:

1. What is the productivity of a team producing a specific program?
2. What is the productivity that one environment (i.e., HPC machine, compiler, development tools) provides over a second environment?

In this paper we will use the first point of view. We want to be able to compare the productivity of two different teams developing the same program on the same hardware platform.

In measuring productivity over the lifetime of a software project, software developers have typically been concerned with the amount of output a development generates for the amount of effort used to produce that output. For commodities (e.g., pencils, telephones, computers, tons of coal) output is easily computed as the number of such objects produced. However, for software the situation is murky. What does it mean for one program that costs $2M to have "more output" than a similar program that costs $1.5M? *Utility* is an economic concept that tries to measure such value. It is a stakeholder's attempt to measure the increased value for an increase in input to the economic system. But computing this increase in utility is very inexact and somewhat subjective.

Therefore, output for software is often measured in the numbers of lines of code, and cost is often expressed as the number of hours required by the staff to produce that output. But as the utility discussion above implies, size is a highly dubious number. What about code quality and other attributes such as maintainability and reliability? With great reliance on COTS (Commercial Off the Shelf) products, use of program libraries, reuse of modules, object oriented class libraries, etc. such calculations are very hard to interpret. Using other measures such as function points doesn't really help. They have the same problem as lines of code in being an imprecise measure of the value of a product.

While lines of code per day (or per hour of effort) have been a traditional measure of productivity – mostly because of the lack of a good alternative – it lacks a true

11th IEEE International Software Metrics Symposium (METRICS 2005)
1530-1435/05 $20.00 © 2005 IEEE

definition of utility for that software. This issue becomes evident in looking at the utility of programs required by the HPC community. They are concerned about measuring and evaluating the performance of the various high end computers used to solve complex problems. For these stakeholders, execution time is often the dominant factor and various benchmarks have been developed for measuring performance on these machines. However, the HPC community is quite aware that there are other critical factors involved in setting up the problem solution: time to program the solution and tune it for maximum efficiency, problem set-up times, times waiting in job queues and scheduling delays, and post execution processing of the results. These times can be significant [5, p. 60].

In this domain, the chief performance measure used is *time to solution*. "The single most relevant metric for high-end system performance is *time to solution* for the specific scientific application of interest." [5, p.6] Very little is known about the various development processes (e.g., how one designs, codes, tests) are used in this environment. In particular, what are appropriate models of productivity for this domain so that various environments can be compared for their effectiveness?

An HPC machine with 1000 processors, for example, has the capacity to execute a program in 1/1000 of the time of a machine with only one processor. The trick is to divide up the problem space so that every processor can be kept busy solving the problem in parallel. If we could do this, then we would achieve a *speedup* of 1000.

Realistically, however, we achieve a speedup much less than this theoretical maximum. While the total processing power increases linearly with the number of processors, it becomes harder to divide some problems into equal-sized parallel pieces. Adams [1] gives an example where a solution that used 92% of the total computational power of 32 processors used only 48% of the total processing power when the machine was expanded to 128 processors.

In developing HPC programs, there is the need to tune a program to utilize the multiple processors of an HPC system. This parallelization effort becomes a significant part of the total cost of the system. Because of this parallelization effort, for this class of stakeholders, utility is related to the speedup an HPC machine can achieve and productivity can be reduced to some function of the execution time of the program and its development time.

We will first discuss several other theoretical models that measure productivity using high performance computers. Can we use these theoretical models in order to develop a computationally effective way to measure programmer productivity in the HPC domain? Just as important, will these productivity models be computationally effective? That is, can we actually run experiments to collect the relevant data to compute this productivity? We ran several experiments using students in high performance courses at several universities, and we will report on those results as it applies to our own model of productivity.

The work reported here is part of the Defense Advanced Research Projects Agency (DARPA) High Productivity Computing Systems (HPCS) program. The mission of the HPCS productivity team is to better understand how one develops software that will execute on the next generation of high performance computers. The HPCS program consists of several Working Groups and the activities presented in this paper represent work that came out of the Development Time Working Group, led by the University of Maryland [4]. As part of developing new high performance machine architectures, research from this HPCS activity may play a role in deciding on the software environments (e.g., compilers, editors, development tools) that need to be built to make those machines most productive.

2. Productivity

2.1. Models of productivity in the HPC domain

In the HPCS program, the primary interest in a productivity metric is to support the acquisition process, as one of several criteria for deciding which system to purchase. Such a metric can also be useful for a programmer or project manager faced with the decision of which parallel programming technology to use for a particular project. Note that in the former case, the metric would be applied to evaluate an entire HPC system, and in the latter case would be applied to evaluate a particular technology on a given HPC system.

A productivity metric can also be applied on an individual project, to evaluate how productive the programmers were in developing and executing the software on a particular HPC system. We restrict our discussion to productivity at the project level. To apply a productivity measure on a technology or system level would involve some method for aggregating the productivity measures from individual projects, which is outside the scope of this paper.

Several models have been proposed to define HPC productivity. Here we present 3 of them as typical of the many that have been proposed. See [7] for additional proposed models.

2.1.1. Utility values

Snir and Bader are one example of using utility theory to the problem of developing a model of productivity [10]. *Cost* is the cost of developing the solution plus the cost of using the system, but the *value* is the utility preference of a stakeholder (e.g., Laboratory director) to the work performed by the system. The utility of a solution will be how long it takes for the solution to be ready relative to the need for using that solution.. Utility, related to the concept of risk, is a function over [0..1] and in this case the utility is generally a decreasing function of time, reaching 0 just at the deadline when the solution *must* be available.

For example, consider the problem of weather prediction. Assume you want to know the weather 48 hours in advance. If you could accurately compute the weather for 2 days in the future using only 1 hour of computation, then the 47 hour advance notice has some economic value (e.g., should you plan an outdoor activity or if it will rain, move indoors?). If it takes 24 hours to compute the weather 2 days hence, then the 1 day advance notice still has value, but less so than if would have after only 1 hour of computation. Finally if it takes more than 48 hours to compute the weather, the calculation has no value; you can simply look out the window to see what the weather is.

As stated earlier, because of the importance of time to solution as a performance metric, the use of utility functions to address this attribute has appeal as a productivity measure. But as a measure of productivity, this approach has two weaknesses:

1. It is somewhat subjective since the utility of the solution over time depends upon the subjective opinion of the stakeholder creating the utility function.
2. Productivity is very dependent upon the particular needs of the application being developed. That is, the time that the solution is needed greatly affects the utility curve that is produced. Therefore, two different programs may have the same performance and development time characteristics, but very different utility curves. This model, while theoretical pleasing, becomes difficult to use in practice.

2.1.2. Work Estimator

Sterling [11] developed an alternative model that defines productivity as utility over cost where utility is the total lifetime work produced by a system and cost is the total lifetime costs of the system, or more formally as:

$$\Psi_w = \frac{S_P \times E \times A}{C_L} \qquad (1)$$

where S_P is the peak performance of the machine, E is the efficiency of the computation (i.e., how much of the peak performance is actually used), and A is the availability of the machine (i.e., how much wall-clock time is actually needed to compute one second of the solution) while C_L represents the total lifecycle costs to develop and execute the program. The numerator represents the calendar time needed to execute a solution to the problem by factoring in how much of the peak performance a computation uses, how efficient the computation is and how much time is spent in various job queues waiting for access to the HPC machine. Note that only execution characteristics in the final solution are used to compute *value* in this calculation of productivity. The time to develop the program is not considered (although the cost to develop is part of C_L).

2.1.3. Power and efficiency

Kennedy [9] looks at the problem from the perspective of the power of a given programming language by defining power and efficiency with respect to a language.

Let P_0 be a standard reference implementation of a program. We can define $I(P_0)$ as the implementation time for P_0. If L is another language we also compute $I(P_L)$ and the ratio of the two, ρ, represents the programming efficiency of implementing a solution in P_L over a solution in P_0 (which he calls power). We can express this as:

$$\rho = \frac{I(P_0)}{I(P_L)} \qquad (2)$$

This gives a value of how efficient it is to program in L over the reference language P_0.

Similarly we can compute the execution efficiency ε as the ratio of execution times, or

$$\varepsilon = \frac{E(P_0)}{E(P_L)} \qquad (3)$$

This ratio shows how much execution time it takes to solve the problem in L relative to the reference language.

Kennedy's time for a solution is the sum of implementation time plus execution time, or:

$$T(P_L) = I(P_L) + r\,E(P_L) \qquad (4)$$

where $T(P_L)$ is the time to solution for program P in language L, and *r* is a problem specific factor giving the relative importance of implementation time over execution time (e.g., *r* is large for programs that execute many times, *r* is small for a program that executes once). Since his goal is to evaluating various languages, he wants to find the P_i that minimizes $T(P_i)$. By choosing an appropriate value for *r,* the impact of I(Pi) becomes more or less important to decide whether only execution time or development time is the more important component of $T(P_i)$.

Substituting for $I(P_L)$ and $E(P_L)$, using the definitions of ρ and ε, we get the time to solution as:

$$T(P_L) = \frac{1}{\rho} I(P_0) + \frac{1}{\varepsilon} r\, E(P_0) \qquad (5)$$

Under the assumption that ρ and ε are relatively constant for a given language L, productivity is then the ratio of the time to solution of P_0 divided by the time to solution of P_L, which gives us the result:

$$\text{Productivity} = \frac{T(P_0)}{T(P_L)} = \frac{\rho + \varepsilon X}{1 + X} \qquad (6)$$

where

$$X = \frac{r\, E(P_L)}{I(P_L)} \qquad (7)$$

This approach has some similarity to the one we adopt later in this paper. However, it has one weakness in its present definition to be applicable within the HPCS program. Often L and the reference language P_0 are variants of the same language and implementation of one aids the implementation of the other. As we later state, a system like MPI simply adds the parallelization primitives to a language like C so this straightforward computation of ρ in equation (2) may be difficult to compute since $I(P_L)$ also includes $I(P_0)$ and it may not be able to obtain an accurate value of $I(P_L)$.

While all of these models compute a value for productivity, our goal was to produce a model we could easily measure in the laboratory and would also be of practical value. Beginning with a simple definition of productivity, we proceed in the next section to develop an alternative view of productivity.

2.2. Traditional productivity

In the HPC domain, productivity, as we stated before, can be represented as:

$$\Psi = \textit{Productivity} = \frac{\textit{Utility}}{\textit{Cost}} \qquad (8)$$

For traditional software projects, this has been an elusive concept. We don't really know the utility of a piece of software, so its size is a reasonable estimator of its complexity and usefulness and the effort to build the product is a reasonable estimator of the cost to produce it. So *Utility* is often expressed as source lines of code (SLOC) and productivity is often expressed as SLOC per hour of effort, and numbers like 10-20 lines per 8-hour day, averaged over the entire development process, are not uncommon (e.g., 160,000 SLOC divided by 20,000 total hours give a productivity of 8 SLOC per hour or 64 SLOC per day).

But if we could quantify program utility (assuming we knew what one unit of program utility represented) we could define productivity accurately. Let M represent the number of program utility units contained in the program. *Traditional productivity* can then be defined as:

$$\textit{Traditional Productivity} = \frac{\textit{Program utility units}}{\textit{Effort in hours}} \qquad (9)$$

2.3. HPC productivity Model

In order to quantify the value of an HPC program, we need to compute its productivity as given by *utility/cost* of equation (8). But what are the utility and cost of such programs? If we assume that we can measure the traditional productivity of an organization (by equation (9)) then the starting point for HPC productivity is this utility in program utility units per hour, which represents the development effort for building a large program.[1] We will assume here that it remains relatively constant for an organization, although we know that it actually can vary greatly from project to project.

Creating a program for an HPC machine has a cost factor generally not present in building a program for a single processor. Often a C or FORTRAN program is written to execute on a single processor, and then a parallel version is built by adding function calls that invoke processes to create parallel paths of execution in separate processors. MPI and OpenMP are two common interfaces used to parallelize a program. The Message Passing Interface (MPI) builds a network of processes that pass messages for communication and OpenMP creates multi-threaded, shared memory parallelism. It requires significant effort to achieve this parallelism by adding the appropriate MPI or OpenMP interfaces to a serial version of the program.

In order to compute productivity in the HPC domain, we need to know the increased value of a program that achieves a speedup of *s*. We make the following two assumptions:

1. **The economic utility is modeled as a combination of program size and speedup.** A program with *M* program utility units (e.g., SLOC) that achieves a speedup of *s* on an HPC machine has the same economic utility as *s* versions of the program, each of size *M* and each running on a single processor. The utility of this code is then *s*M*. That is, achieving a speedup of *s* is equivalent to *s* implementations of the program, each of size *M*.
2. **The cost is modeled as development effort.** Because of the complexity of achieving parallelism, an HPC

[1] For ease in understanding, you can substitute SLOC for program utility units. As stated previously, SLOC is the most commonly used proxy for these units.

program has an increased cost of development. Let *C* represent the relative increase of producing the parallel version of the program over the single processor serial version. *C* can be expressed as the ratio of the cost of the parallel version over the cost of the single processor version. and for HPC programs *C* is generally greater than 1.

Let *k* be the cost for producing 1 program utility unit. If *M* is the program size in program utility units, then the uniprocessor cost is *M*k*. The cost of the parallel program is then *M*k* program utility units times the relative cost factor of *C* or *M*k*C*.

Combining both assumptions, we get a tentative HPC productivity measure as:

$$\Psi = \frac{s*M}{M*k*C} = \frac{s}{k*C} \quad (10)$$

or

$$\Psi = \frac{1}{k} * \frac{s}{C} \quad (11)$$

where *s* is the speedup and *C* is the relative cost or effort for creating and parallelizing the program (e.g., total effort to both create the serial and then parallel version of the program). The constant *k* represents the costs to produce 1 program utility unit of the final program (or as stated previously, can be approximated by the cost to produce one line of code). Instead of program utility units we can use any proxy for that figure. So it is irrelevant if we use hours of effort, SLOCs, function points, or some other attribute as long as this attribute measures utility in some direct way.

Equation (10) gives us an absolute value for productivity, but unfortunately we have no idea what *k* and a program utility unit are. However, given two different developments, if we divide one productivity value by the other, the *1/k* factors cancel, and we can use equation (11) to compute the relative productivity between two different projects. In this case, if we let P be the HPC program we wish to evaluate and P_0 be a reference implementation of the same program, then P/P_0 provides a relative value for parallel implementation P compared to the reference implementation P_0. For a reference implementation we choose to use a baseline serial implementation (i.e., an implementation running on one processor). In most cases we will set the serial execution time *s* as 1 and the relative cost *C* as 1, so baseline serial productivity will be $\Psi = 1/k * s/C = 1/k$. In this case, dividing the two productivity values gives us a measure of utility of the parallel system over relative costs compared to a serial implementation.

We can then represent HPC productivity, by the following model, originally developed by Jeremy Kepner of MIT Lincoln Labs (and Project Director of the HPCS productivity team) that we have been using for HPC programs. We define productivity as given in Figure 1.

$$\text{Speedup} = \frac{\text{Serial Execution Time}}{\text{Parallel Execution Time}}$$

$$\text{Productivity} = \frac{\text{Relative Speedup}}{\text{Relative Effort}}$$

Figure 1. HPC Productivity Model.

2.3.1. Relative Speedup

The numerator of our HPC productivity model, *relative speedup,* sounds like a simple concept, but has significant issues when trying to compute a reasonable value. For one thing we would like to use relative speedup as a means to compare two different implementations of a program, perhaps to compare the effectiveness of the underlying language or parallelization model (e.g., MPI versus OpenMP). However, one can artificially inflate the relative speedup by using a poorly implemented serial version. For example, one can build a serial implementation, measure the serial execution time, then simply turn on compiler optimization switches and recompile. One can often achieve a speedup of 2 or 3 that way with no parallelization effort at all. (i.e., Relative effort has a value of 1, with a resulting productivity of 2 or 3.)

A more serious problem is that it is often not easy to even get the serial implementation. Many developers work on both the serial and parallel implementations at the same time and the project may not even pass through a development stage where there is a complete serial implementation to measure. Some problems are simply too complex to test or run via a serial implementation on a single processor.

In order to get a level playing field, one solution to this problem is to use a reference serial implementation R_1. It actually doesn't matter which one is used since we are only comparing relative HPC productivities. That is, using a different reference implementation R_2 will have the effect of changing all values of productivity by the factor R_1/ R_2 and the relative ranking of each project will remain the same.

Obtaining a reference serial implementation also has the effect of validating the value of the program in the time to solution equations. If we can obtain an optimal serial implementation, then a parallel implementation is superior to this serial implementation only if it has a productivity

greater than 1 according to Figure 1. That is, we need a speedup sufficient to counteract the increased effort of the parallel implementation.

2.3.2. Relative Effort

For a similar reason, we can often not even obtain both the serial and parallel effort for a development since parallelization may occur during the initial development of the solution. However, if we substitute the effort for building the serial reference implementation, as with relative speedup, we get uniform value we can use (up to a multiplicative constant).

Combining both of these results, we get an equation for HPC productivity as given in Figure 2. Note that our equation for relative effort is a variant of the value ρ from Kennedy's model given in equation (2) and speedup is simply Kennedy's efficiency ε from equation (3). The difference is that in Kennedy's case he is using as a reference implementation a standard parallel solution to the problem. In our case, we want the reference implementation to be a serial implementation.

$$\text{Speedup} = \frac{\text{Reference Execution Time}}{\text{Parallel ExecutionTime}}$$

$$\text{Productivity} = \frac{\text{Relative Speedup}}{\text{Relative Effort}} = \frac{\rho}{1/\varepsilon} = \rho \times \varepsilon$$

$$\text{Relative Effort} = \frac{\text{Parallel Effort}}{\text{Reference Effort}}$$

Figure 2. Revised HPC productivity

In order to provide normalization to the values of ε and ρ, a good choice for this reference implementation should be the best serial implementation in terms of execution time that can be written. As stated at the end of Section 2.3.1, ρ represents the additional effort needed to parallelize all solutions to the problem over a good serial implementation, and ε represents the ultimate speedup one can achieve over the reference implementation. However, if no optimal implementation is available, then any serial implementation will do, except that the values of ρ and ε will have to be interpreted differently.

3. Experimentation

As part of our work on the HPCS program, we have been evaluating development characteristics for HPC programs by studying the development of programs within selected high performance computing classes at several universities [4]. Each semester 2 or 3 graduate HPC-related classes at several universities run several experiments addressing HPCS development time issues. These experiments are done within the context of the programming assignments for the course. Typically students develop 2 or 3 programs using several standard models (e.g., MPI, OpenMP) and we collect all compilation and execution time characteristics for these programs as well as copies of all source programs compiled by the students. In addition the students (are supposed to) turn in logs giving the time spent on performing the various tasks in completing the assignment. From this data we are able to evaluate the various workflows used to build those programs. We can use this data in order to evaluate our model of productivity described in this paper.

$$\text{Speedup} = \frac{\text{Reference Execution Time}}{\text{Parallel ExecutionTime}}$$

$$\text{Productivity} = \frac{\text{Relative Speedup}}{\text{Relative Effort}}$$

$$\text{Relative Effort} = \frac{\text{Parallel SLOC}}{\text{Reference SLOC}}$$

Figure 3. Simple HPC productivity measure

Accurate effort data is usually very difficult hard to obtain [6]. Programmers are either reluctant or find it difficult to keep accurate records on their activities. However, we have already shown by equation (11) that any measure of relative effort suffices. As long as the application domains are similar, SLOC counts are an approximation of effort, so within a limited domain, Figure 3 gives an easily computed formula for productivity. We use this as well as effort in our classroom experiment.

3.1. Evaluating productivity measures

Table 1 presents an initial evaluation of the productivity measures from Figures 2 and 3. These represent 5 students in one class where we have the relevant data in solving Conway's game of life program in C using MPI as the parallelization interface.

Part A represents the calculations where actual effort data is used to compute productivity. Program 3 was rated as the highest productivity. The program with shortest execution time (Program 5 with an execution time of 8.5 seconds) was only the second best in terms of productivity. Even though it took 4 seconds longer to run (about 50%), total effort to construct the program was only 10 hours compared to 22 for program 5. Program 4 which also ran

about 10% faster than program 3 took 34.5 hours to build instead of the 10 hours of program 3. The other 2 programs clearly had lower productivity both in terms of effort to complete and in performance.

Table 1. Productivity – Game of Life program					
Part A. Using effort for productivity					
Program →	**1**	**2***	**3**	**4**	**5**
Serial effort (hrs)	3	7	5	15	
Total effort (hrs)	16	29	10	34.5	22
Serial Exec (sec)	123.2	75.2	101.5	80.1	31.1
Parallel Exec (sec)	47.7	15.8	12.8	11.2	8.5
Speedup	1.58	4.76	5.87	6.71	8.90
Relative Effort	2.29	4.14	1.43	4.93	3.14
Productivity	0.69	1.15	4.11	1.36	2.83
Part B. Using SLOC as effort measure					
Serial SLOC		161			134
Parallel SLOC	123	352	300	1827	151
Relative SLOC	0.76	2.19	1.86	11.35	0.94
SLOC Productivity	2.07	2.18	3.15	0.59	9.49
*- Reference serial implementation					

Using SLOC counts to estimate effort, we get the results in Part B of Table 1. We believe the results are similar, but not as accurate. The two programs with highest productivity are the same and the bottom 3 are the same. In this case program 5 does have the highest productivity, due principally to the extreme brevity of the MPI source program compared to program 3.

Applying this analysis to a second program, we get the results described by Table 2. This second problem was the Buffon-Laplace needle problem, a Monte Carlo simulation, where the value of π can be computed by dropping a needle randomly on a grid. In this case effort data is mostly missing and several serial executions are not available. But using the SLOC counts and parallel execution, we can get a representative productivity value.

In this example, the reference implementation chosen was program 18. It had the optimal serial execution time at 3.17 seconds, but exhibited an all too common phenomenon in the HPC domain that the parallel version running on 8 processors actually ran slower at 3.42 seconds (i.e., a speedup of less than 1).

Program 11 was judged highest in productivity at 5.81 with program 15 at 4.01 not far behind. Program 11 indeed has the least execution time at 0.70 seconds with program 15 being the second at 0.94. Program 11 also had the characteristic of requiring the least lines of code at 43 with program 15 only requiring 46.

Tables 1 and 2 also demonstrate some of the practical problems in computing productivity measures. We did not obtain the serial effort for program 5, but fortunately it is not needed in our calculations of Ψ. Similarly, serial SLOC counts for programs 1, 3 and 4 were also missing. Almost all effort data for programs 11 through 22 was not submitted by the students. Fortunately, again these are not needed to compute our productivity measures. As we've said previously, obtaining intermediate values (e.g., serial effort, serial SLOC count, serial execution times) are very hard to obtain and basing our calculations on final values greatly increases the usefulness of the formulas, even if the precision is not as great. All we need is some reference implementation upon which to base our relative measures.

3.2. Discussion

The productivity measure has an implicit time to solution component built in with the calculations of ε and ρ. But going back to Table 1, do we want to say that program 3 with productivity of 4.11 is always better than program 5 with productivity 2.83? If this is a problem executed once (what is often called the "lone researcher" workflow), then program 3 probably is better since we are trading off 12 development hours for a decrease of 4.3 seconds of execution time. But if this is a problem that will

Table 2. Productivity – Buffon Laplace needle problem												
Part A. Using effort for productivity												
Program →	**11**	**12**	**13**	**14**	**15**	**16**	**17**	**18***	**19**	**20**	**21**	**22**
Serial effort (hrs)							4		4			12
Total effort (hrs)									7			
Serial Exec (sec)	3.33	5.10	20.00				6.20	3.17	3.90			5.91
Parallel Exec (sec)	0.70	1.07	2.50	2.97	0.94	4.00	1.95	3.42	1.50	1.98	1.81	1.81
Speedup	4.54	2.96	1.27	1.07	3.36	0.79	1.62	0.93	2.11	1.60	1.75	1.75
Relative Effort	---	---	---	---	---	---	---	---	---	---	---	---
Productivity	---	---	---	---	---	---	---	---	---	---	---	---
Part B. Using SLOC as effort measure												
Serial SLOC	32	40	28		36	37	50	55	33			43
Parallel SLOC	43	52	43	50	46	57	66	92	55	62	42	77
Relative SLOC	0.78	0.95	0.78	0.91	0.84	1.04	1.20	1.67	1.00	1.13	0.76	1.40
SLOC Productivity	5.81	3.13	1.62	1.17	4.01	0.76	1.35	0.55	2.11	1.42	2.29	1.25
*- Reference serial implementation												

run daily for the next 10 years, then we want to increase the importance of execution time over effort in computing productivity. Much like the factor *r* in Kennedy's time to solution equation (4), we need to modify productivity to reflect the execution versus implementation tradeoffs in computing productivity. In the next section we offer some comments on approaching that problem.

4. Productivity revisited

The productivity formula of Figure 3 has the weakness that it doesn't address the total lifetime execution behavior of an HPC program. In order to address that factor, we propose to modify the equation by reducing the impact of the cost factor (e.g., relative effort term in Figure 3) when a program is repeatedly executed. As in Kennedy's productivity model, we propose a factor *r* in the calculation of relative effort to cause this effect, as in:

$$\text{RelativeEffort} = \frac{(\text{ParallelEffort} + \text{r} \times \text{Reference Effort})}{(\text{Reference Effort} + \text{r} \times \text{ParallelEffort})} \qquad (12)$$

The value *r* represents the number of executions of the program over its lifetime and varies between 0 and 1. This is only a temporary solution. As we show, this modification has most of the properties we want, but we need to establish it on a better theoretical foundation.

For the single execution lone researcher type of program, *r* has the value of 0 and equation (12) reduces to Figure 3 with productivity given by Tables 1 and 2. However, if the program represents one used repeatedly, then *r* is close to 1 and the *Relative Effort* value approaches 1, meaning that the influence of development time is less on productivity and the utility of the program reduces to just speedup.

Table 3. Kennedy's productivity from Table 1

r	Prg. 1	Prg. 2	Prg. 3	Prg. 4	Prg. 5
1	0.44	0.24	**0.70**	0.20	0.32
100	0.52	0.31	**0.88**	0.26	0.41
1000	0.95	0.84	**2.06**	0.74	1.14
10000	1.45	2.96	**4.74**	3.29	4.73
20000	1.51	3.64	5.24	4.39	**6.14**

Returning to the example presented in Table 1, there was a question of whether program 3 or 5 had higher productivity. Program 5 had higher speedup of 8.90 over 5.87 for program 3, but required 12 more hours of development. Our analysis chose 3 as the better program.

In Figure 4 we plot the values of *Productivity* for both program 3 and program 5 as *r* varies between 0 and 1. The solid line represents program 3 and the dashed line represents program 5. For *r*=0 we get the productivity of Table 1 and for *r*=1 we get productivity as simply the speedup of the various solutions with program 5 being most productive. At *r*=0.26 the lines cross, changing the answer of which program has higher productivity.

Since parallel effort (consisting of the total time to write a serial then parallel version of the program) is so much greater than the serial effort alone in the HPC domain, if we assume that the parallel effort is greater than the reference effort, *Relative effort* becomes a monotonically decreasing function of *r,* so productivity is monotonically increasing. Because of this, we can always eliminate solutions with lower speedup values. So for Table 1, programs 1 and 2 can never have highest productivity for any value of r. Program 4, with speedup of 6.71 potentially has higher productivity than program 3, but by plotting program 4 (dotted line) we see that program 3 and program 4 have the same productivity of 5.59 at *r*=0.76, but program 5 already is higher at 7.73.

In order to better resolve this issue, we still need to evaluate equation (12) so that:

1. We have a better understanding of the modifications of relative effort to better reflect the contribution of development time over the repeated execution of the program over its lifetime.
2. We need to correlate the factor *r* to the number of repeated executions of a program. (That is, what does *r*=0.26 mean for the example graphed in Figure 4? What is the physical reality expressed by r=0.26?)

As a final check on our model, we compare our productivity results with the results from Kennedy's productivity analysis (Equation (6)). Table 3 presents his productivity results for the data in Table 1 based upon the value of *r* chosen. We use the same program 2 as the reference P_0 of his analysis.

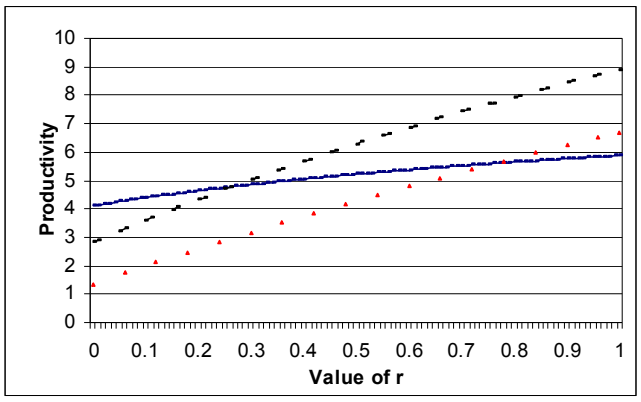

Figure 4. Productivity for programs 3, 4, and 5.

We get a result similar to his model. For *r*=1, program 3 has the highest productivity (in bold in Table 3). But as *r* increases, the execution component of T(P) becomes more important and for r=10,000 program 3 and program 5 have

almost the same productivity. For larger *r*, program 5 has the highest productivity, demonstrating that for large number of executions, implementation time is less important than execution time speedup. (As with our model, Kennedy needs a correlation of the value of *r* with a measure of total execution time.) In both models, the limiting value in productivity as execution time increases will be the speedup factor.

Table 4. Rank of each program by each model

r	Model	1	2	3	4	5
small	Kennedy	2	4	1	5	3
	Our model	5	4	1	3	2
large	Kennedy	5	4	2	3	1
	Our model	5	4	1	3	2
Very large	Both	5	4	3	2	1

Table 4 compares the productivity ranking of the programs in the two models for increasing values of *r*. (Remember that *r* is an integer for Kennedy and a number between 0 and 1 for our model.) Both identify the same highest ranking program for small *r*. In our case program 1 received the lowest ranking because of its extremely high execution time compared to the others. In Kennedy's case all 5 programs were ranked the same as their total development effort. For our data at least, execution time did not play a role in his model until the influence of execution time (via increasing *r*) was significant.

As *r* increases the two rankings converge. Both will have the same rankings for extremely large *r*, being just the speedup factor.

5. Conclusions

In this paper we have addressed the issues of productivity in the high end computing domain. Although measures such as source lines of code per day or function points per day are traditionally used in the software engineering world as measures of productivity, these measures are not applicable in all domains. In the HPC world, time to solution counts as well or may be even more important.

We have been working on a productivity model that uses relative speedup and relative costs as the primary drivers of the model. What still needs to be developed is a relationship of the parameters of our model and real world characteristics of HPC programs. Can we use our productivity measure to create an absolute measure of productivity rather than simply a relative measure among a set of implementations? This requires an understanding of the constant *1/k* we dropped from equation (11). While *k* was not important in the analysis performed here, it is necessary in order to compute an absolute value for productivity. Also, the factor *r* introduced in Section 4 needs a relationship with physical reality and measures the relative costs of development time versus repeated executions.

In our HPCS project, we have been investigating productivity via a series of experiments in student classes. Our initial productivity measure, given by Figure 2 seems appropriate and its modification using SLOC counts in Figure 3 gives results that are generally available and easy to compute. Looking at the effects of repeatedly executing a given program with a long lifetime, the effects of development time diminish in importance. We therefore get the tentative absolute productivity formula given in Figure 5. (We put back the constant *k* we dropped previously in evaluating the class experiments in Section 3.) (While actual effort is the preferred measure, SLOC counts instead of actual effort may also be used. In this case, *k* would be the cost to produce 1 line of code.) As given in the previous section, we need to better understand and evaluate the *Relative Effort* term in this formula.

$$\text{Speedup} = \frac{\text{Reference Execution Time}}{\text{Parallel ExecutionTime}}$$

$$\text{Productivity} = \frac{\text{Relative Speedup}}{k \times \text{Relative Effort}}$$

$$\text{Relative Effort} = \frac{(\text{Parallel Effort} + r \times \text{Reference Effort})}{(\text{Reference Effort} + r \times \text{Parallel Effort})}$$

Figure 5. Modified Productivity

The major weakness in using the formula in Figure 5 is in capturing actual effort data. SLOC counts do not give as accurate a result. This data has been difficult to capture accurately in our experiments, and from 30 years experience with industrial projects (such as the NASA Software Engineering Laboratory), it is just as difficult to capture accurately in industry [3]. This is something we are well aware of and have been addressing it. In our experiments, effort time is captured in 3 ways:

1. **Student logs**. This is the method reported in this paper and the one we believe is most accurate, when it can be collected. Students turn in web-based forms periodically on the effort spent on various activities.
2. **Instrumented compilers**. We have implemented scripts that students use to compile and execute their programs. These capture machine times quite accurately. However, it doesn't capture think time when the student is thinking about the problem or in evaluating changes to make in the source program

while editing the program. This is a significant part of the total effort.

3. **Instrumented environment**. We have also been using Johnson's Hackystat system to automatically extract information from the computer system as students work [8]. Hackystat works by plugging in to an editor, listening for edit events, and recording the activity the student is working on and what file is being manipulated. This gives an indication of what computer resources are being used, but doesn't address the same "think time" problem of the instrumented compiler solution.

For now we rely mostly on student logs, but are investigating ways to automate the process better. For Tables 1 and 2 we chose those classroom projects where we believed we had the most accurate effort data. One idea we have been looking at is integrating Hackystat with the Eclipse environment [2] to make the process more transparent to the student and we hope more accurate. This and other approaches are being studied.

The major impact of this work is the realization that productivity may have different meanings in different application domains. Source lines of code per day works in traditional software development environments, but for other applications, such as the HPC domain discussed here, we need to go back to the original economic definition of productivity as value divided by cost and define each of these in the context of that environment.

Acknowledgement

This research was supported in part by Department of Energy contract DE-FG02-04ER25633 to the University of Maryland. We wish to acknowledge the contributions of the various faculty members and their students who have participated in the various experiments we have run over the past 2 years. This includes Alan Edelman at MIT, John Gilbert at the University of California Santa Barbara, Mary Hall at the University of Southern California, Alan Snavely at the University of California San Diego, and Uzi Vishkin at the University of Maryland. We are also indebted to Jeremy Kepner of MIT Lincoln Labs for his contributions to the basic productivity model.

References

[1] M. F. Adams, H. H. Bayraktar, T. M. Keaveny, and P. Papadopoulos, Ultrascalable Implicit Finite Element Analyses in Solid Mechanics with over a Half a Billion Degrees of Freedom" SC2004, Pittsburgh, PA, November 2004.

[2] J. Arthorne and C. Laffra, Official Eclipse 3.0 FAQs Addison Wesley Professional, 2004.

[3] Basili V., F. McGarry, R. Pajerski, M. Zelkowitz, Lessons learned from 25 years of process improvement: The rise and fall of the NASA Software Engineering Laboratory, IEEE Computer Society and ACM International Conf. on Soft. Eng., Orlando FL, May 2002, 69-79.

[4] Carver J., S. Asgari, V. Basili, L. Hochstein, J. K. Hollingsworth, F. Shull, M. Zelkowitz, Studying Code Development for High Performance Computing: The HPCS Program, Workshop on High Productivity Computing, ICSE, Edinburgh, Scotland, (May, 2004) 32-36.

[5] Workshop on The Roadmap for the Revitalization of High-End Computing, Computing Research Association, June 2003.

[6] Hochstein L., V. Basili, M. Zelkowitz, J. Hollingsworth and J. Carver, Combining self-reported and automatic data to improve programming effort measurement, Joint 10th European Software Engineering Conference and 13th ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE 2005), Lisbon, Portugal, September 2005.

[7] International Journal of High Performance Computing Applications (18)4, Winter 2004.

[8] P. M. Johnson, H. Kou, J. M. Agustin, Q. Zhang, A. Kagawa and T. Yamashita, Practical automated process and product metric collection and analysis in a classroom setting: Lessons learned from Hackystat-UH, Proceedings of the 2004 International Symposium on Empirical Software Engineering, Los Angeles, California, August, 2004.

[9] K. Kennedy, C. Koelbel, and R. Schreiber, Defining and measuring the productivity of programming languages, The International Journal of High Performance Computing Applications, (18)4, Winter 2004, 441-448..

[10] M. Snir and D. A. Bader, A framework for measuring supercomputer productivity, The International Journal of High Performance Computing Applications, (18)4, Winter 2004, 417-432.

[11] T. Sterling, Productivity metrics and models for high performance computing, The International Journal of High Performance Computing Applications, (18)4, Winter 2004, 433-440.