

Productivity Metrics and Models for High Performance Computing

Thomas Sterling[†]
California Institute of Technology

April 20, 2004

1. Introduction

A conceptual framework is required for understanding the relative merits and utility of alternate computer systems for high performance: their architecture, execution model, and software methods for programming and operation. The general concept of “*productivity*” has been articulated as that attribute of the entire process of computing that delivers ultimate value to the end user mission. Productivity is defined to reflect the rate, value, and costs of producing computational results in contribution to achieving mission objectives of the using institution. While there is general consensus as to the importance and abstract nature of a productivity value of merit, there is not a common viewpoint of the specific metric and its formulation by which to quantitatively characterize it. The contribution of this paper is to propose a conceptual framework within which to consider productivity as a parameter and a formulation by which to quantify it. The objective of this framework is to establish a set of mutually consistent and complementary metrics that satisfy the discipline of dimensional analysis and that together provide a rigorous definition of productivity. It is the intent of this work that its results may serve as a tool for evaluation and comparison of alternate high performance computing systems.

For purposes of this paper, three closely related terms are adapted to distinguish among the degrees of quantitative certainty that can be assigned to properties of a system or process.

- **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,
- **Metric** is a parameter that is observable and measurable.

In describing various aspects of computing that contribute to productivity, the distinction among these three properties enables the broadest range of consideration while ensuring a rigorous foundation for at least part of its definition. For example, it is understood that programming languages are an

[†] This research was carried out at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. The funding for this research was provided for by the Defense Advanced Research Projects Agency under task order number 15506, under the NASA prime contract number NAS7-1407.

important factor in software productivity. But there is no clear quantitative means of representing language. Reliability is a quantifiable parameter but it cannot be directly measured. Statistical means of estimating maximum likelihood exist but the actual value of its lifetime cannot be determined beyond post mortem (and it's a little too late by then). On the other hand, the actual performance of a computer system can be measured and treated as an explicit metric defining its operation; at least as is related to a specific application, compiler, etc.

The process to which the concept of productivity is being applied is one that involves both the execution of an application program to produce a sought after result, and the construction of that application program. Productivity can be measured either for the machine performing the computational work or for the team developing the applications and ultimately employing their results. Figure 1 depicts the basic process by which results are generated with a high-end

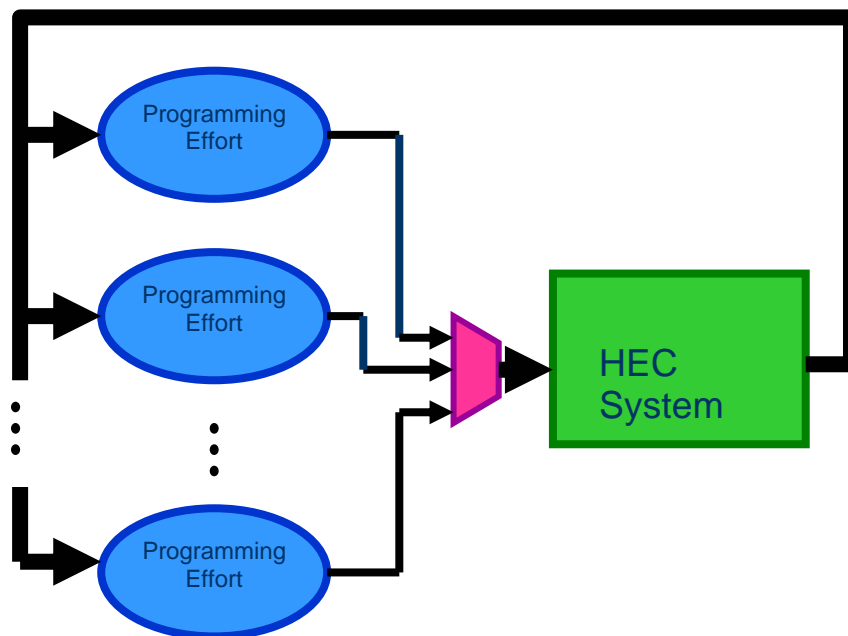


Figure 1. Development and Execution Cycle

computer (HEC). The system is operated at its maximum throughput while the applications performed on it are constructed separately and potentially concurrently.

The model of *machine productivity* can be construed as a throughput process while the model of *team productivity* can be represented as a serial response time process. In the first case, the critical time path is that related to the machine capability, including its inefficiencies, with the assumption that there are always user programs to be performed. In the second case, the critical time path is the

sum of an individual programming effort and the response time of the machine performing that single application, including the waiting time for access to the machine. To make any progress with these concepts, with the ultimate goal of establishing a set of interrelated parameters and their combining formula relationships, we need to define “*productivity*.”

The purpose of this paper is to present one possible formulation of the concept of computational productivity through a series of thought experiments and derivations. The resulting terms, parameters, and equations are not asserted as either the only or best codification of productivity but only as a concrete and self-consistent example of what properties a final representation might embody. One constraint imposed on any such formulation is that it satisfies the discipline of rigorous dimensional analysis. Every parameter must have a unit assigned to it that is consistent with the units of the parameters and their combining relationship that define it.

This paper employs two representation schema in conjunction with a methodology on incremental refinement to expose and codify the principle factors and their interrelationships that define productivity. Section 2 presents a general model of productivity from which all following concepts are derived and conform. This model is a simple algebraic relation that introduces the concept of the result as the basis for productivity without making the specific units of results explicit. Rather, this is left unbound in order to achieve the power of flexibility. In Section 3, a graph representation (the 2nd schema) is introduced to reflect a hierarchy of factors and their qualitative associations.

This paper was inspired by Burton Smith, Chief Scientist of Cray Inc. and the PI of the Cascade Project, and motivated by a requirement of Bob Graybill of DARPA IPTO, the program manager of the High Productivity Computing Systems program. Burton invented the concept of the PFDG (see Section 3) to represent the complex interrelationships among the many factors that determine a system’s productivity and the ways in which the Cascade concepts contribute significantly to realizing these factors. Bob has challenged the HPCS program to engage in a strategy of evaluation based on quantifiable and repeatable metrics to maintain the highest standards of research. It was he who requested/required that each vendor participating in Phase 1 of the HPCS program address the question of evaluation metrics for productivity. Since the release of the first tentative model (version 1) at the DARPA Scottsdale meeting in October of 2002, a number of colleagues in the community have taken the time to assess and improve upon it. This paper (version 4) benefits directly from these contributions and the author wishes to acknowledge with gratitude the contributions of Richard Games and David Koester of Mitre, Jeremy Kepner of MIT Lincoln Laboratory, Jeff Vetter of Lawrence Livermore National Laboratory, Ed Upchurch of JPL, and from Cray Inc.: Brad Chamberlain, David Callahan, Allan Porterfield, and Keith Shields. Also, Marc Snir of the University of Illinois has developed a related

model that has influenced this work as well, and is reflected in a companion paper.

2. A General Model of Productivity

We define “productivity”, Ψ , to be that property of the complete computing process that is the measure of the rate of computing weighted-results per unit cost. The product that is produced is the result, R , (or sequence of results), a non-negative real-valued parameter. A result is a complete solution to a computational problem represented by a set of result data; the answers to the computational problem. One could treat all the results as equal (all results are created equal) and simply quantify the total production of a machine during its lifetime as the number of results produced. But in most cases, this characterization is inadequate to fully capture the variability of the degree and impact of results to the mission objectives to which the machine (and programmers) are dedicated and against which the ultimate benefits are evaluated. As an analogy, compare two assembly lines: one produces toasters and the other automobiles. The toasters and automobiles are the products or individual results of their respective assembly processes. The toaster assembly line might produce toasters at a greater rate than does the automobile assembly line produce automobiles. But in most cases, we would equate the result of a new automobile to be of greater worth than the result of a new toaster (you may reflect on possible exceptions). The total production of a machine during its lifetime, R_L , is the weighted sum of all of the results produced by the machine.

$$R_L = \sum_i^{N_R} R_i$$

where N_R is the number of results produced during the lifetime of the machine and R_i is value attributed to the i^{th} result. Therefore, the rate of production over the lifetime, T_L , of the machine is the ratio of the sum of the weighted values of the results and the time required to produce them.

$$\text{rate of production} = R_L/T_L$$

During the lifetime, T_L , of the machine, it may transition repeatedly among several states. These include the total time, T_R , to produce the useful results, the total time, T_V , spent performing all of the overhead functions of the machine, and the total down time, T_Q (quiescent) during the lifetime of the machine. If T_i is the time required to produce the i^{th} result R_i , then

$$T_L = T_R + T_V + T_Q$$

and

$$T_R = \sum_i^{N_R} T_i$$

A second normalizing factor is required: the cost of production during the lifetime of the machine, C_L . Two lines of production may produce the same product value in the same lifetime. But if the first production line costs substantially more than the second, the rate of return on investment for the second would be much greater than the first: its productivity would be higher. Alternatively if one were to spend as much for the second as the first, then the second line would be scaled up to produce more (although not necessarily linearly proportional) during its lifetime for the same cost; again exhibiting a higher productivity in the form of return on investment. The cost during the machine lifetime must reflect several aspects of the process of computing. These include the procurement and initial installation cost of the machine, C_M (which in extreme cases may include the cost of the building in which to put the machine such as the Earth Simulator), the total cost of operating the machine, C_O , and the cost of constructing the application software for producing all of the results, C_S . In this relation, we are integrating over the entire lifetime of the machine. Therefore C_S is the cost of construction of all of the application software for the entire lifetime of the machine. Note that this does not include system software. This is covered under the procurement and operational costs, C_M and C_O , respectively. For future use, two related coefficients are defined related to cost. The amortized cost of the machine per unit time, c_m , is related to C_M by:

$$C_M = c_m \times T_R$$

The per time unit cost of operating the machine, c_o , is related to C_O by:

$$C_O = c_o \times T_R$$

These are defined in terms of T_R rather than T_L because it is assumed that only the useful time on the machine will be paid for and the costs must be amortized across the users. The total cost of application software construction for the life of the machine is the sum of the cost of application construction for each result:

$$C_S = \sum_i^{N_R} C_{S_i}$$

and thus,

$$C_L = C_S + C_M + C_O$$

Finally, these three parameters are combined in to a single relationship to derive a general model for productivity, Ψ . It is cast in terms of the entire lifetime of the machine.

$$\Psi_L = \frac{R_L}{C_L \times T_L}$$

For a span of time, t , less than the lifetime of the machine, a slightly different formulation is required where the costs are amortized across a subset of the total machine lifetime and the products considered are only those results accomplished during the designated time frame.

$$\Psi_t = \frac{R_t}{C_t \times t}$$

While the discussion has been posed in terms of machine productivity, the arguments and final formulation of the general model for team productivity is the same. However, the costs, results, and time will be different, even for the same system over the same time interval.

3. Productivity Prime Factors

From the general model presented in the previous section, it is possible to begin to delineate the basic factors that determine the degree of productivity being achieved by the entire computing process including hardware system, software support, and application development. As a means of representing the hierarchy of candidate factors determining the ultimate productivity of a computing process, we establish the *productivity factors directed graph* (PFDG). By convention, the factors to the left are of a higher order, in essence accumulating the contributions of the lesser (more detailed) factors to the right while acquiring their units where appropriate. Ideally, every factor would embody one or more parameters with specific units. While this will not always be possible in our full model, their effects will be quantifiable, if not directly measurable. For example, the factor of language in its broadest sense, while very important to software productivity, is itself not quantifiable. However, its effects in terms of units of time and cost of application program construction can be quantified, if not measured. As an initial, albeit incomplete, abstract representation of productivity factors, Figure 3.1 introduces a partial PFDG as a simple sparse three-tiered tree:

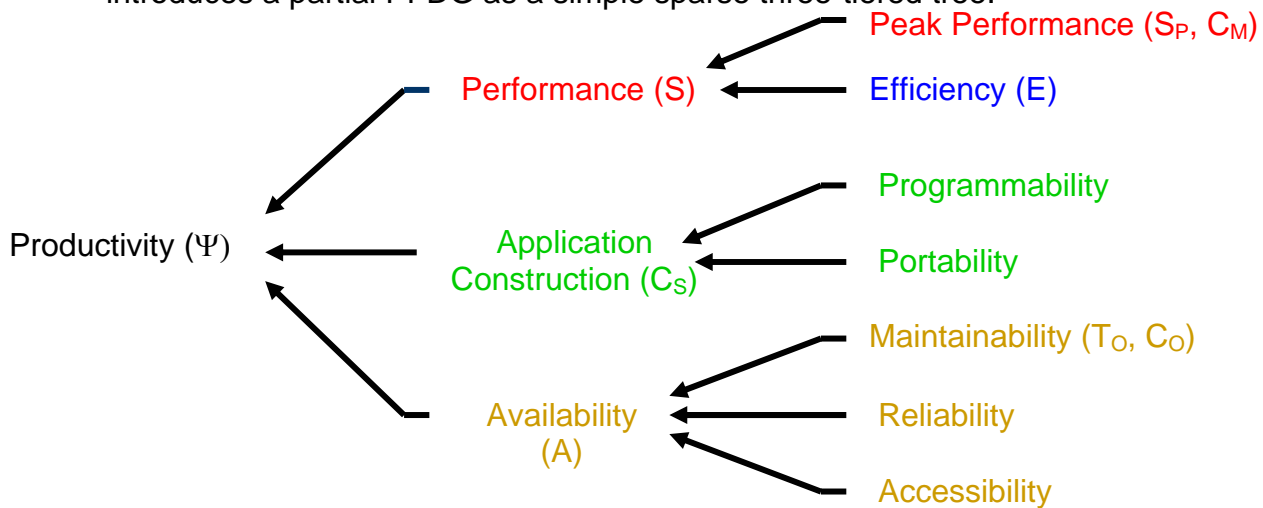


Figure 3.1 – Productivity Factors Directed Graph

Without asserting completeness, it is possible to reason about the factors that contribute to productivity within constraints of correctness and dimensional analysis. Two strategic issues may be determined immediately when considering a candidate factor: does it directly influence its immediate superior factor (the one it points to on the left) and is that superior factor directly or indirectly proportional to the candidate factor?

Productivity at the left is symbolized by the variable Ψ and from the previous section holds the units of result-units per cost-time. Productivity is directly proportional to *performance* (S), which is a parameter with units of compute operations per unit time. Performance is the sustained or average throughput achieved on the user applications that produce the ultimate results; higher performance delivers more results during the lifetime of the machine. Performance is directly proportional to the parameters of *peak performance* (S_P) also measured in units of compute operations per unit time and *efficiency* (E), which is the fraction of sustained to peak performance.

$$S = S_P \times E$$

Productivity is significantly impacted by the factor referred to here as application-construction. Application construction is the cost (C_S) of developing an application code that, in turn, delivers one or more results. It is also the time (T_S) to develop the application codes. For the machine model, only the cost contributes explicitly with the time a subfactor in determining the cost. For the team model, both the cost and the time contribute directly, i.e. the cost of application code development contributes to the total cost of productivity and the time to develop the application code contributes to the total time of productivity. We will return to the details of application construction in a later section but here assert that two factors of significance to application construction are programmability and portability. Programmability is the relative difficulty with which new code components may be developed by a user-programmer. Portability is the relative difficulty with which old code components may be modified and applied by a user-programmer.

$$C_{Si} = c_f \times T_{Si}$$

where c_f is a constant of proportionality in units of FTE (full time equivalents) and the subscript i refers to an individual application result.

Productivity is directly proportional to the fraction of the machine time that is available to execute user applications. The parameter is *availability* (A) and is the ratio of total user time and lifetime of the machine.

$$A = T_R/T_L$$

Availability is a consequence of several factors including the machine's *reliability*, how long it takes to fix the machine when it breaks or during scheduled maintenance (its *maintainability*), and the contention for access to the machine when it is operational. For the machine model, this last factor, *accessibility*, reflects the overhead work such as operating system services that takes time away from user time on the machine. For the team model, accessibility also involves the conflict experienced by a team for use of the machine caused by sharing the machine with other application teams. This is equivalent to the waiting time in a queue.

While this first PFDG presents many of the dominant factors contributing to productivity, it does not fully define productivity because it does not establish what the unit of results is in a quantifiable terms. In fact, as such a unit must reflect the value of benefit to the end user team or agency mission, there is no single unit that will satisfy all cases. Therefore, the general model is intentionally left open with no final binding on this question. Instead, it is anticipated that a series of special models will be derived from the general model that incorporate specific meaning to the result factor and bind its quantifiable units. The next section gives an example of such a special model using total work (number of operations performed) as an estimator of quantitative worth of having accomplished the computed result.

4. A Special Model of Productivity with Work Estimator

The value of the product or result, R_i , must be asserted to derive a specific and quantitative measure of productivity. One possible estimator of the value of the results produced by a computing system (and its programmers) during its lifetime is the total amount of useful work, W_{RL} , performed throughout the execution of the application programs. Useful work can be estimated as the number of basic operations performed to carry out the calculations of the application algorithm and distinguished from ancillary overhead work, W_{VL} , required to manage parallel resources and concurrent tasks, and from duplicated work, W_{DL} , that is the same calculation performed at different locations to avoid undue global communication and synchronization. Thus the total work, W_L , performed during the execution of user applications is:

$$W_L = W_{RL} + W_{VL} + W_{DL}$$

and,

$$R_L \cong W_{RL} = \sum_i^{NR} W_{Ri} = \sum_i^{NR} \{W_i - (W_{Vi} + W_{Di})\}$$

where W_i is the total work (number of operations) performed for the i^{th} application. The work based special model or w-model is an estimator of productivity given by:

$$\Psi_w = \frac{W_{RL}}{C_L \times T_L}$$

Work as an estimator for the value of results has a strong attraction because it is an intuitively satisfying indicator and provides a true metric, a measurable parameter with explicit units. But even in this narrow case, it can only be approximate as closer examination exposes its limitations and inconsistencies. Not all operations are created equal. It is not unreasonable to assess a floating-point operation such as divide or square root with a greater value of intrinsic worth than the bit-by-bit logical *OR* operation. Yet, here all operations are enumerated with the same weight. Not all programs for the same end-user application perform the same number of operations. Worse, again for the same application, a worse program, that is one written to do too much work, is given a better value than a well-written program that requires a minimum amount of work for that application. This is also true when different implementations of math libraries are integrated as part of the user program. Different computers have somewhat different instruction sets. Two different computers running exactly the same source-code application program will inevitably produce at least a slightly different operation count. This variation is likely to be aggravated by the use of different compilers and compiler optimizations. Thus the metric of operation count, W , is at best an approximation of a class of worth attributed to a given result, R .

For a given application, we define efficiency, E_i , as the fraction of peak performance (maximum operations capable of being performed in unit time), S_P , producing useful work, W_{Ri} , in time T_i . E_i takes in to consideration not only the inefficiencies of the processors' execution pipelines but also the overhead and duplicate work occurring during the application execution.

$$W_{Ri} = S_P \times E_i \times T_i$$

and

$$W_{RL} = \sum_i^{N_R} (S_P \times E_i \times T_i) = S_P \times \sum_i^{N_R} (E_i \times T_i)$$

An average efficiency over the lifetime of the machine, E , is given by:

$$E = \frac{\sum_i^{N_R} (E_i \times T_i)}{T_R}$$

$$W_{RL} = S_P \times E \times T_R$$

so that,

$$\Psi_w = \frac{S_P \times E \times T_R}{C_L \times T_L}$$

and remembering that availability A was given as:

$$A = T_R/T_L$$

then a new formulation for the w -model of productivity is:

$$\Psi_w = \frac{S_P \times E \times A}{C_L}$$

5. Software Productivity

The total cost, C_L , of developing the results, R_L , over the lifetime of the machine is the sum of the costs of procuring (C_M), maintaining, and operating (C_O) the machine and the costs of construction of the end-user application programs (C_S) as discussed in Section 2.

$$C_L = C_S + C_M + C_O$$

The cost of software development is proportional to the level of effort or aggregate time spent constructing the end user applications with a constant of proportionality, c_f , equivalent to the FTE or full time equivalent of a canonical programmer (this in spite of the fact that the ability of programmers to write code varies by as much as an order of magnitude as does their salaries; unfortunately, these two are not tightly correlated) so that:

$$C_S = c_f \times T_S$$

where T_S is the total level of effort to develop all of the application programs performed during the lifetime of a machine. There have been many attempts to model the effectiveness of programmers in developing software. Here, we take a somewhat simplistic approach by delineating the separate stages of constructing an application, measuring the size of the partition of the program associated with each stage, and determining the rate at which each stage can be developed.

It is asserted here, without proof, that the construction of a program involves three distinct styles of code development, depending on the initial condition or starting point of the code to be created. In the most trivial case, the application program already exists and is already tuned for the particular machine of choice. All that is required is configuring the code to accept the input and output data set files, run on the specified sub partitioning of the target machine, and interface

with possibly other code segments with which it is to interoperate. Most common among this *reuse* category are user codes previously written by the same team and common or shared libraries that have been tuned for the target platform. The other extreme is constructing an application or some part of it entirely from scratch. This category of *new* codes requires the development of the algorithm as well as the actual writing of all new lines of code and can be the most difficult and time consuming. Between these two extremes is the adoption of existing codes by modifying them, possibly extensively, to satisfy the computing requirements and operate effectively on the target platform. This *porting* category is usually associated with transferring a program that runs well on one machine on to another machine and attempting to make it run well there too. While this is certainly part of porting, it is the minor part. Many if not most codes are ported not between machines but between functionality on the same machine. Wherever possible, a programmer will modify an old code that has some similarities to the new functionality desired, changing the minimum amount of the original code required. This is a very productive approach to building new applications and has been codified as a class of methodologies referred to as *templates*, a form of codes intended to be custom modified to fit specific functional requirements and execution platforms.

For each of these three categories, we define the parameter ρ_x to be the relative partition or fraction of the complete application program that is devised through that method so that ρ_{new} represents the fraction of the total application that is constructed from scratch, and so on. The amount of time required by the canonical programmer to construct a unit sized segment of code for each of these categories is represented by the parameter τ_x such that, for example, τ_{port} is the time to port unit-sized code. The development of an application program also involves debugging both for correctness and performance optimization. This level of effort is also codified by a time measure, τ_{bugx} , designated for each of the three categories of programming such that the time of debugging programs that are being reused is represented by τ_{buguse} . Finally, all six of these code-size normalized time measures of program construction or debugging are in units of time per unit code-size. The size of the application program to be constructed is represented by the symbol, L_i , and is in arbitrary units of code-size. We do not define this unit further at this time, recognizing that many employ *SLOC* or software lines of code for this purpose.

To organize these parameters, into a more convenient form, four vectors of three elements each are constructed from the original scalar parameters. The vector for the time per code size unit of programming for a given application is:

$$\bar{\tau}_{prog,i} = [\tau_{new,i}, \tau_{port,i}, \tau_{use,i}]$$

The vector for the time per unit code for debugging a given application is:

$$\bar{\tau}_{bug,i} = [\tau_{bugnew,i}, \tau_{bugport,i}, \tau_{buguse,i}]$$

And the vector for the fraction of each partition of the total program size is:

$$\bar{\rho}_i = [\rho_{new,i}, \rho_{port,i}, \rho_{use,i}]$$

Because the programming and debugging times for a given program are of the same units, they can be combined in a element-wise vector sum such that:

$$\bar{\tau}_i = \bar{\tau}_{prog,i} + \bar{\tau}_{bug,i}$$

The time to construct an application is proportional to the inner product of the rate and partition vectors:

$$T_S = \Gamma_i \times (\bar{\tau}_i \bullet \bar{\rho}_i)$$

Therefore the cost of constructing a given application is:

$$C_{Si} = c_f \times \Gamma_i \times (\bar{\tau}_i \bullet \bar{\rho}_i)$$

and the cost of application construction for the lifetime of the machine is:

$$C_S = c_f \times \sum_i^{NR} \{ \Gamma_i \times (\bar{\tau}_i \bullet \bar{\rho}_i) \}$$

Applying this to the earlier relationship for the w-model of productivity:

$$\Psi_W = \frac{S_P \times E \times A}{c_f \times \sum_i^{NR} \{ \Gamma_i \times (\bar{\tau}_i \bullet \bar{\rho}_i) \} + C_M + C_O}$$

Although correct in principal, this measure of productivity requires that the entire lifetime of the machine be treated in ensemble while one is more likely to consider a narrower epoch of time, perhaps as short as the interval required to execute a single application. But there arises a subtle question as to what is the time span over which the work accomplished for a single application is rated against. If it were simply the period required to execute the program, it would be inconsistent so it must be prorated by the availability as follows. Note that here we have been a bit sloppy, dropping off the subscripts and assuming that the variables refer to a particular application and its execution.

$$\Psi_w = \frac{S_P \times E \times A}{c_f \times \left\{ \Gamma \times (\bar{\tau} \bullet \bar{\rho}) \right\} + (c_m + c_o) \times T}$$

6. Concluding Remarks

The model of productivity offered here demonstrates an approach that permits reasoning about productivity in a quantitative and consistent methodology. Only the machine oriented model was presented. The team model is derived similarly and will be presented in a further paper. Also, many of the parameters used in this relationship are themselves products of more detailed formulation. Efficiency is one important example. This is a factor that can be characterized in terms of sub parameters including latency, contention, overhead, and starvation, thus delineating the explicit causes of performance degradation. Value was treated in the special model as proportional to work performed. Some of the limitations of this predictor were already discussed. But another aspect of this issue, well described by Snir in his paper and further articulated by Burton Smith in closed discussions is that the value of an application execution is limited in a nonlinear way by the amount of time it takes to be performed. It is asserted that beyond a certain bounding time, the problem result is of no value. This important consideration needs to be integrated in to the model to capture time to completion as a determining attribute of result value. As was pointed out by a reviewer of an earlier draft of this paper, the “uniqueness” of HEC systems is not exposed through this formulation in the sense that capability computing is not shown to be superior to systems employed primarily as capacity (or throughput) engines. Both the team model and the finite bounded response time constraints will better represent this attribute. The cost of programming, while correct in a dimensional sense, is unsatisfying in that these parameters are really averages and not easily observed. A more tractable formulation of programming cost may be required. Nonetheless, in spite of these limitations, the proposed model meets its initial goals and allows us to reason about productivity with a good foundation.

7. References

Bailey, D.H. Twelve ways to fool the masses when giving performance results on parallel computers. *Supercomputer Review*, 4(8):54–55, 1991.

Bailey, D. H., Barszcz, E., Barton, J., Browning, D., Carter, R., Dagum, L., Fatoohi, R., Fineberg, S., Frederickson, P., Lasinski, T., Schreiber, R., Simon, H., Venkatakrisnan, V., and Weeratunga, S. The NAS Parallel Benchmarks. Technical Report RNR-94-007, NASA Ames Research Center, (March 1994).

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.

High Productivity Computing Systems (HPCS) Program,
<http://www.darpa.mil/ipto/Programs/hpcs/index.htm>

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).

McCalpin, J.D., STREAM: Sustainable Memory Bandwidth in High Performance Computers.

Numrich, R.W., Performance Metrics Based on Computational Action. 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).

Smith, B., Personal Communication, (March 2002).

Snively, A., et al. A Framework for Performance Modeling and Prediction. In SC2002. 2202. Baltimore, MD.

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).

TOP500 Supercomputer Sites, <http://www.top500.org/>