Measuring HPC Productivity*

Stuart Faulk
University of Oregon
faulk@cs.uoregon.edu

John Gustafson
Sun Microsystems Inc.
John.gustafson@sun.com

Philip Johnson
University of Hawaii
johnson@hawaii.edu

Adam Porter
University of Maryland
aporter@cs.umd.edu

Walter Tichy
University of Karlsruhe
tichy@ira.uka.de

Lawrence Votta
Sun Microsystems Inc.
lawrence.votta@sun.com

Abstract
One key to improving high-performance computing (HPC) productivity is finding better ways to measure it. We define productivity in terms of mission goals, i.e., greater productivity means that more science is accomplished with less cost and effort. Traditional software productivity metrics and computing benchmarks have proven inadequate for assessing or predicting such end-to-end productivity. In this paper we introduce a new approach to measuring productivity in HPC applications that addresses both development time and execution time. Our goal is to develop a public repository of effective productivity benchmarks that anyone in the HPC community can apply to assess or predict productivity.

1 Introduction

While computer performance has improved dramatically, real productivity in terms of the science accomplished with these ever-faster machines has not kept pace. Indeed, scientists are finding it increasingly costly and time consuming to write, port, or rewrite their software to take advantage of the new hardware. While machine performance remains a critical productivity driver for high-performance computing applications, software development time increasingly dominates hardware speed as the primary productivity bottleneck. Traditional benchmarks do not provide a way to measure productivity.

Removing or ameliorating productivity bottlenecks in next-generation high-performance computing systems is a key objective of DARPA's High-Productivity Computing Systems (HPCS) Program. This objective has been characterized as the long-term goal of improving software productivity "at a rate commensurate with improvements in the underlying hardware" [Kepner 04].

Stating the goal this way reflects the scientific community's experience that improved hardware performance often does not yield similar improvements in important measures like total cost, effort, or time to solution. Addressing these larger productivity issues will require addressing productivity bottlenecks in software development, project management, and system administration in addition to hardware performance.

* This work is sponsored in part by the Defense Advanced Research Projects Agency (DARPA) under Agreement No. NBCH3039002.

Our work focuses on developing an effective approach to characterizing and measuring HPCS software productivity. Before we can hope to address software productivity problems, we must agree on what those problems are and how we would determine whether we had solved them. In short, we must agree on what we mean by "productivity" in HPCS applications, and how such productivity should be measured, before we can rationally address the problem of how best to improve it. Goals of the work include:

1) *Develop a common definition of HPCS productivity* that the HPCS developers, suppliers, and buyers (e.g., government agencies) can agree on. For our purposes, this means a definition that is consistent with the mission-level view that greater productivity means that more science is accomplished with less cost and effort.
2) *Develop effective measures of HPCS productivity* that encompass the overall development process—design time as well as execution time. In particular, we seek to develop measures that apply to a wide range of development environments and broadly across high-performance computing application domains (e.g., weather prediction, fluid dynamics, nuclear applications, mechanical design, etc.) to assess, compare, or predict productivity.
3) *Provide productivity measurement capabilities* to guide productivity improvement for both hardware and software developers. Effective, objective measurement provides the basis for systematic productivity improvement. We seek to provide common, public benchmarks and metrics to use in assessing and improving productivity.

Both HPCS developers and buyers have traditionally used standardized benchmarks (e.g., LINPACK) to guide development choices. System developers use the benchmark results to guide platform development and subsequently demonstrate the speed of their machines. Buyers traditionally use such benchmarks to predict computation times and choose among competing platforms. However, the benchmarks and corresponding metrics employed to date have proven to be decreasingly effective predictors of end-to-end productivity. Traditional benchmarks focus almost entirely on hardware speed. Thus, they typically attempt to predict only execution-time productivity, ignoring development time. Further, they do not measure other properties of an application that matter to users: reliability, repeatability, portability, reusability, maintainability, etc.

In this paper we describe our approach to creating a new type of standardized benchmark that 1) encompasses the breadth of design-time and execution time activities as well as 2) the productivity contributions of both functional and non-functional requirements. In addition to defining a canonical computation problem, these "productivity benchmarks" seek to characterize an end-to-end productivity problem by capturing the representative context of the computation. This includes the overall process in terms of canonical workflows [Kepner 04] as well as the developmental attributes that contribute to the overall value of the software to its stakeholders. We will call such a multi-dimensional productivity benchmark a *productivity benchmark suite* (PBS).

Briefly, a PBS comprises a canonical problem in the context of a complete set of behavioral and developmental requirements representative of a particular high-performance computing domain. In addition to the functional and non-functional requirements, the productivity benchmark suite (PBS) will provide targeted metrics and

tools for measuring productivity in terms of overall costs and benefits across the development cycle. The goal is to create a set of benchmarking capabilities that, when applied, will exercise and measure not only the execution efficiency of a platform on a particular class of high-performance computing problems, but all the dimensions of development that contribute to the value of a solution.

Our long-range goal is to develop a public repository of well-validated PBSs that are representative of the productivity challenges in each distinct high-performance computing domain. Platform developers or buyers can apply these PBSs to assess and predict productivity of particular high-performance computing platforms on their domain interest. In particular:

1) HPCS Buyers: buyers of HPCS platforms are faced with the problem of predicting which vendor's system will provide the best "bang for the buck" in their application domain. This question embraces the total cost of ownership including software development, execution, operation, maintenance, and so on. Buyers will be able to apply benchmarks from the repository to answer specific questions about how different platforms or development strategies can be expected to affect their productivity.
2) HPCS Developers: HPCS platform vendors seek to develop systems that will improve productivity for their customer base. To do this, they must understand what the productivity problems are in their customer's application domains, what properties of the platform affect those productivity issues, and how to measure the results of platform design changes. Vendors will be able to apply benchmarks to guide architectural design.

A goal of our work is to provide the HPCS community with the capabilities needed to determine whether new technologies will effectively address critical productivity problems before vendors deploy those technologies in next-generation platforms. This will reduce the risk that next-generation high-performance systems will fail to meet DARPA's HPCS productivity goals.

Our overall approach is empirical in the sense that we will develop PBSs based on observations of real developers and their productivity problems with actual high-performance applications. We will ensure validity of the benchmarks and metrics through carefully controlled experiments.

The following sections describe the planned approach and expected results in detail. First, we describe a framework for reasoning about HPCS productivity and illustrate its application with a real application. We then describe ongoing work in developing a framework for PBSs and our approach to productivity measurement. We illustrate a new approach to measuring a system's developmental qualities (e.g., maintainability). Finally, we discuss ongoing work in creating and validating productivity benchmarks.

2 Reasoning about HPCS Productivity

The economic definition of productivity is the **output per unit-of-work**. For example, "worker productivity" denotes the value of goods and services produced in a period divided by the hours of labor used to produce them. While the units vary depending on

the realm of discourse, all definitions of productivity share the underlying concept that increasing productivity means producing more of value with less work.

Conceptually, the economic definition of productivity is both simple and intuitively appealing. It is also consistent with the objectives of the HPCS program – i.e., providing platforms that allow scientists to accomplish more science in less time at lower cost. Unfortunately, the economic definition has proven difficult to apply to software development. This is true not just for high-performance computing applications, but for the broad classes of software applications more typically addressed in Software Engineering productivity studies (e.g. [Boehm 80] and [Boehm 81]).

The mutable, intangible nature of both the processes and the products of software development make the **outputs** or **units-of-work** difficult to define or measure. The default has been to choose metrics that are relatively easy to measure, but that bear only a loose relationship to the value of what is produced.

Our goal is to address these issues by providing a framework for characterizing and measuring the perceived *value* of the output to system stakeholders. We define the output to include any properties of the system that consume work and have stakeholder value, including those that have no direct physical analog in the code (e.g., usability).

The remainder of this section describes the conceptual framework for our approach. We first discuss some of the historical difficulties and pitfalls associated with software productivity metrics. We then describe a general framework for modeling a software development's output value based on the stakeholder values of developmental properties and requirements. We argue that the proposed framework is sufficiently flexible to address productivity issues across development time and run time. We then walk through an example taken from requirements for a high-performance application that illustrates consistency of the model with a real high-performance computing application.

2.1 Pitfalls of Traditional Productivity Measures

Software productivity has been and remains a core mission of software engineering. Nonetheless, problems in adequately measuring software productivity, much less predicting or improving it, have proven among the most intractable in the field. While there is general agreement that software productivity should reflect the economic definition: **output** value per **unit-of-work**, there is little agreement on how to define the outputs or the units of work. In general, industry experience has been that those software development properties that we can most easily and precisely measure (e.g., the number of lines of code produced) have little direct relationship to the system's stakeholder value while many properties with established value (e.g., maintainability) have no established, effective metric.

The long-running controversy over the most common software productivity metric, the number of source lines of code developed (SLOC), illustrates the types of problems that arise. SLOC became one of the first commonly accepted productivity metrics [Boehm 81] and remains in regular use [Boehm 95]. However, if applied incautiously, SLOC introduces a number of uncertainties and potential paradoxes. The number of lines of code necessary to implement a particular functionality will vary greatly from one programming language to another. Worse, they can vary inversely with the quality of the programmer and efficiency of the code (i.e., better programmers do more with less code).

If used without careful controls, SLOC productivity measures may indicate that productivity has decreased when it has actually increased [Jones 86]. In short, creating lines of code is a necessary but not sufficient condition to creating software of value. There is no predictable relationship between SLOC (or similar metrics like. function points [Albrecht 83]) to critical code properties that enhance value like maintainability, portability, reusability, security, and so on.

Paradoxes and uncertainty likewise attend the use of common **units-of-work** like labor hours or, more typically in software, man-months. Frederick Brooks' first articulated some of the problems in using conventional units of work to measure or predict software productivity in his book [Brooks 95], *The Mythical Man Month*, most famously, his observation that, adding manpower to a late software project typically made it later. Subsequent empirical software engineering research at NASA Goddard's Software Engineering Laboratory (e.g., [Porter 95]) underscored the diversity of factors that affect the cost and quality of software production. Likewise, detailed time-motion studies of software developers [Perry 96] suggest that even the people doing the work do not accurately predict or even recall exactly what they spend their time on. Such work indicates that making reasonably accurate correlations between units-of-work and produced value requires careful, detailed empirical study of the development process in a controlled context.

Similar issues arise with productivity measures in high-performance computing. Traditionally, measures of machine performance like the peak number of floating point operations per second (FLOPS) are used as a predictive measure of high-performance computer output. Historically, such measures had validity where the total time to solution was overwhelmingly constrained by program execution time, and where the program execution time was overwhelmingly dominated by the time spent doing floating-point operations. As the effort needed for developing, porting, rewriting, and tuning the software has increased, the relevance of such measures has correspondingly decreased. (Floating-point arithmetic also no longer dominates execution time even in the most compute-intensive programs, yet the focus on FLOPS persists.) Likewise, as program properties other than execution performance (e.g., portability, maintainability, etc.) have become more important production values, the need for metrics that account for these properties has increased as well.

Early productivity studies observed the phenomenon that Weinberg and Schulman [Weinberg 74] characterized as "what-you-measure-is-what-you-get" (WYMIWYG). Repeated studies have consistently confirmed this observation's predictive power. If one measures productivity in terms of the number of lines of code, programmers will produce reams of code. If one measures HPCS productivity in terms of FLOPS, then we will get processors that show very high arithmetic rates on carefully chosen benchmarks. In neither case will we necessarily get any improved productivity in the sense of more functions implemented or more science done per dollar spent.

An important corollary is that the reverse also holds true. In short, if we want to get something, we should measure it. This principle dictates, in particular, that if we want developmental properties like portability, interoperability, maintainability, etc. in addition to execution time properties like performance and accuracy, then we need to measure all of those properties as directly as possible. Even if we ignore past pitfalls, this principle

suggests that we cannot use a single metric that obscures the productivity contribution of each critical property. Rather, we will need measures that explicitly address development output as comprising multiple, diverse properties of value.

2.2 An HPCS Productivity Framework

Intuitively, each application development has associated with it some value to its stakeholders. The overall value will typically be due to a number of different properties of the code, its execution behavior, and its development process. Exactly which properties have value, and how much value is attributed to them will vary from one type of application to the next and from one class of stakeholder to the next. In particular, we observe that:

1) The overall value is a function of a number of distinct properties of the static work products, the code, and the development process itself. In addition to accuracy and timeliness of results, these may include run-time properties like security, availability, and locality. It may also include desirable developmental properties like portability, maintainability, or reusability or even organizational concerns like consistency with organizational standards or the amount of legacy code reused.
2) Exactly which attributes of the system are important will vary from one type of application to the next, from one class of stakeholders to another, and possibly even from one run of the application to another.
3) The relative importance or priority of each attribute may likewise vary by application type, stakeholder class, and so on.
4) The relative values of individual properties as well as the total value of a given solution may change over (calendar) time [Snir 04].

In short, the value associated with any particular development is typically a function of a number of different properties that we can expect to vary from one development effort to the next or even one run of the program to another. We can capture this by representing the total relative value as a vector over the values of the properties of interest using the following framework. We begin by associating with each property of interest i:

1) A metric of completion C_i
2) A relative value weight v_i

Briefly, the metric of completion C_i is a measure of the degree to which the realization of the property i meets stakeholder goals for that property. The relative value weight v_i represents the importance of the property i relative to the other properties of interest. Assuming independence, the value of some set of properties $i = 1$ to n is given by the vector:

$$V_A = (v_1C_1, v_2C_2, \ldots, v_nC_n) \quad (1)$$

Where we can normalize each of the v_iC_i to a common metric (e.g., labor or cost), we can express the total value as the sum.

$$V_A = v_1C_1 + v_2C_2 + \ldots + v_nC_n \quad (2)$$

For example, we could calibrate each C_i such that $C_i = 1$ whenever property i meets its design goals, and v_i gives the relative importance of property i expressed as a percentage such that $\sum^{n}_{i=1}(v_i) = 100$ and $V_A = 100$ exactly when all the C_i are satisfied. Relative productivity is then given by the value produced divided by the work consumed to produce it:

$$P = V_A / W \tag{3}$$

Equivalently, we can say that the greater the value of V_A for a given amount of work, the higher the productivity. This corresponds to our intuitive view that greater productivity implies greater value per unit of work.

Both equation (1) and equation (2) are useful. Equation (1) has the advantage that the individual contributions of each property of interest to the total value are clear and that we need not attempt to normalize over different kinds of properties. However, equation (2) has the advantage of summarizing the overall value in a way that might be compared across projects if normalized over a common metric like cost.

We anticipate further refinement to our equations to reflect significant dependencies. In the general case, the degree of completeness and value of one system property or even a set of properties may depend on others. For example, it does not make sense to talk about the "value" of properties like maintainability or portability if the code does not do what it is supposed to. Further, the values of individual properties, as well as the program as a whole, are likely to change over time. A solution today is typically worth more than the same solution tomorrow. Thus, the value weight v_i associated with each property i may be a function of calendar time t as can V_A itself. Understanding the nature of these relationships, their significance, and how best to represent them will require careful study of different classes of HPCS applications.

2.3 Applicability of the Value Model

By design, our value function must be used in the context of a computing application that establishes the value space of interest. For productivity benchmarking, this context will be given by the PBS. The definition of the PBS will include the definitions of the properties of interest, corresponding metrics of completion, and representative value weights. Appropriate properties and values will be obtained from empirical studies of representative development efforts in the application area of the benchmark; i.e., for a benchmark simulating behavior of a weather code, representative properties, values, and completion criteria would be gleaned from the weather simulation community. This framework will then be tailored to the intended use of the benchmark. If particular properties are irrelevant, they can be omitted (equivalently, given zero values).

The basis for constructing such productivity models as well as the applicability of our framework can be illustrated by considering an example from a real high-performance application. Tables 1 and 2 are taken from Software Development Plan (SDP) for Virtual Prototyping and Accelerated Testing of DoD Composite Material Combat Systems (VPATC) [VPATC 03] and are part of the definition of system requirements and constraints.

Table 1 gives a subset of the Critical Operational Issues (COI) and Measures of Effectiveness and Suitability (MOE&S) defined for the VPATC system. Table 2 defines what are called the system's Critical Technical Parameters (CTP). It maps the technical parameter values that should be measured back to MOE&S in Table 1 and gives both the minimal and optimal criteria for the CTPs for the scheduled sequences system testing milestones. This sequence begins with the system acceptance test (SAT), follows with the Alpha and Beta tests, and finishes with the initial operational test and evaluation (IOT&E).

Clearly, the value of the VPATC system is not a function of its execution performance alone or even of its computational behavior as a whole. A given implementation of the system requirements will be acceptable only if it meets the minimum criteria for the Critical Operations Issues as detailed in Table 2. For example, a system will be acceptable only if its execution performance meets the criteria that "Fixed speedup exceeds 60% of optimum on 32 or more processors" and the code meets the portability requirement of "running on three HPC platforms with the same valid results." Thus, we observe that:

1) To satisfy requirements, the system must meet development goals for a number of distinct properties *concurrently* with functional requirements.
2) The properties of interest span development and execution time. They include (static) developmental properties (e.g., COI #4: Maintainability and Adaptability), execution time properties (e.g., COI #1: Performance), and organizational properties (e.g., COI #5: Training and Technology Transfer).
3) Associated with each property is some *testable* metric of completion. For example, portability is measured based on the number of HPC platforms that will yield valid results when running the same code.
4) Notions of value, while implicit, are present in rudimentary form. For example, it is clear that an implementation of the system that meets the "Optimum Objectives" listed in Table 2 has greater value than an implementation that meets only "Minimum Objectives." A rough value scale could be constructed based on the number of objectives satisfying optimum criteria.

Table 1: Critical Operational Issues (COIs) and Measures of Effectiveness and Suitability (MOE&S)

COI	COI Title	COI Description	MOE&S
1	Performance	Does the project provide computational results that are accurate, stable, and reliable in a portable scalable environment?	Scalable performance improvement over current systems Software robustness
2	Interoperability	Does the project application code's derived data integrate with reusable software components and scientific visualization techniques?	Sharing of project developed software resources among local and remote users
3	SOS Portfolio Interoperability	Does the underlying framework enable the exchange of results with other simulations?	Sharing of project developed software simulation results in real-time among System-of-Systems simulation users

COI	COI Title	COI Description	MOE&S
4	Maintainability and Adaptability	Does the project software adhere to standards and accepted practices, utilize standard languages and libraries, utilize common visualization tools, and provide adequate documentation?	Portable project developed software products across existing and future HPC platforms Reliability of application software products Maintainability of application software products
5	Training and Technology Transfer	Does the project software reside in a catalogued repository, utilize standard languages, and provide adequate documentation?	Software dissemination and technology transfer
6	Usability	Does the project software's human-computer interface support ease of learning, ease of use, effectiveness and efficiency, and user satisfaction?	Maintainability of application software products
7	Security	Are appropriate access controls in place to safeguard the intellectual property rights and security concerns associated with the project software?	Security

Table 2: Critical Technical Parameters

CTP	Test Event	Evaluation Optimum Objectives	Evaluation Minimum Criteria
Scalable software suites: Demonstrate reduction in clock time as a function of increased Central Processing Units (CPU)	SAT	Determine the effective software code architecture for construction of scalable composite material predictors and dynamic models	Same
	Alpha	Fixed speedup exceeds 50% of optimum on 16 processors	Fixed speedup exceeds 40% of optimum on 8 processors
	Beta	Fixed speedup exceeds 60% of optimum on 32 processors	Fixed speedup exceeds 50% of optimum on 16 processors
	IOT&E	Fixed speedup exceeds 70% of optimum on 64 processors	Fixed speedup exceeds 60% of optimum on 32+ processors
Portable, reusable application software: Software applications behave the same and produce similar results, within an acceptable margin of error, on a variety of scalable HPC platforms	SAT	Architecture of code determined, and approaches for parallel execution analyzed	Same
	Alpha	Codes run on two HPC platforms with valid results	Same
	Beta	Codes run on three HPC platforms with same valid results	Codes run on two HPC platforms with same valid results

CTP	Test Event	Evaluation Optimum Objectives	Evaluation Minimum Criteria
	IOT&E	Codes run on four or more HPC platforms with same valid results	Codes run on three HPC platforms with same valid results
Portable, reusable application software:	SAT	Reusable software components identified.	Determine the architecture for pre- and post-processing
Code provides for data output	Alpha	Data output in Tecplot format and XDMF format for custom interface.	Data output in Tecplot format.
	Beta	Data output in Tecplot format and XDMF format for custom interface.	Data output in Tecplot format.
	IOT&E	Data output in Tecplot format and XDMF format for custom interface.	Same.
Stable, accurate and robust software:	SAT	Determine the architecture for pre- and post-processing	Same
Interface with all required external software products and codes	Alpha	Software stores data in XML/HDF format and supplies interfaces to PETSc and the Scalable Parallel Direct Solver Library for Sparse Symmetric Positive Definite Systems (PSPASES)	Software stores data in XML/HDF format and supplies interfaces to the Portable Extensible Toolkit for Scientific Computation (PETSC)
	Beta	Software stores data in XML/HDF format and supplies interfaces to PETSc and PSPASES	Same
	IOT&E	" "	Same

Clearly, we do not expect that every system specification will express properties and values so clearly or can be as easily mapped to our productivity model. The VPATC specification is, in our experience, unusual in its clarity and specificity, representative of current best practices. Nonetheless, the format and placeholders for COIs and MOE&S reflect government standards and are required for all similar DoD development. Clearly most codes will have similar kinds of requirements though it may take more detective work (e.g., interviews and observation) to characterize them as precisely.

3 Developing a Productivity Benchmark Suite

We observe that reasoning about or measuring productivity in the VPATC domain requires considering a wide range of different kinds of system properties. Clearly, a development effort that meets the optimum criteria for the same effort, time, and cost as one that meets the minimum criteria would be considered more productive. Thus, these

properties should be considered part of the "output" that one must measure to assess productivity.

The same reasoning would apply to developing an effective benchmark for assessing productivity in VPATC's application domain. If, for example, we wanted to use a productivity benchmark to answer a question of the form "Will we achieve greater productivity on a VPATC application using HPC platform A or platform B?" then the benchmark must incorporate the kinds of properties, values and metrics we observe in the VPATC specification. In addition to defining a computational problem that exercises the hardware in the same manner the VPATC simulation does, the benchmark would need to define analogous requirements for the critical system properties (COIs) including interoperability, usability, security, maintainability, adaptability, and so on. Fully executing the benchmark would require solving the computational problem in a manner that satisfies all of these functional and non-functional requirements. Measuring productivity against the benchmark would require measuring the extent to which each of the requirements had been satisfied against the effort expended.

Notionally, this characterizes the content and use of a productivity benchmark suite. We view a PBS as a publicly available package that effectively represents the development challenges characteristic of a particular high-performance application domain. Each such package defines a computational problem in a context that simulates the characteristics and constraints of a typical application in a particular high-performance computing domain. The challenge is to define the context sufficiently that:

1) The context adequately characterizes value space of the end-to-end requirements and goals of a real application domain
2) It is possible to make meaningful comparisons in productivity measures between distinct applications of the benchmark

Our approach is based on the empirical derivation of canonical workflows [Kepner 04] and purpose-based benchmarks [Gustafson 04]. Together with the associated non-functional requirements, value function, and metrics, these sufficiently constrain the problem that different developers should be able to apply the benchmark and generate productivity measures that can be meaningfully compared. This will provide a basis for a public repository of commonly applicable HPCS benchmarks. The key components are:

Canonical Workflows: Briefly, canonical workflows are used to characterize and constrain the process context of a productivity benchmark. A canonical workflow characterizes both the development process and the execution workflow associated with creating and using a high-performance computing application to meet an overall set of mission goals. It characterizes the process steps and work products associated with characteristic development paradigms in the high-performance computing community.

Purpose-based benchmarks (PBB): PBBs are described in detail in a separate article in this issue [Gustafson 04]. Briefly, PBBs are computational problems that accurately embody the design and execution time challenges of real applications in a domain. Unlike traditional benchmarks, PBBs are designed to exercise both the development process and the development platform in essentially the same manner (with reduced size) that real development problems do in a particular application domain.

Non-functional requirements: The benchmark will include representative execution time and developmental requirements with their associated metrics of completion and effectiveness. These include any requirements on the development process, administration, static-design, and run-time behavior characteristic of the application domain.

Characteristic value function: Associated with the requirements is a representative value function (e.g., in the form of the value function (1) or (2) above). The value function characterizes a value proposition (i.e., relative values of the different requirements) associated with applications in the domain interest.

Productivity metrics and tools: A set of standardized metrics, algorithms and tools for measuring productivity associated with both development time and execution time activities and goals. These are discussed further in subsequent sections of this paper.

Our approach to building PBSs is empirical in the sense of being derived either from observation of real developers or from carefully controlled experiments. For example, we will obtain the application properties of interest and their relative value directly from developers in particular HPC domains like by direct inquiry or by observation. The empirical methods we plan to apply to particular parts of the problem are described in context.

3.1 Measuring Development

There are two major goals of empirical measurement in the context of our benchmarks:

1. Characterization: Initially, the primary goal of measuring the processes and products of high productivity computing system applications is to better understand what actually happens during such development. Measurement supports identification of potential problems and bottlenecks in HPCS development, clarification of the similarities and differences between the various workflows for development, and the potential creation of predictive models for required resources and product quality.
2. Control: Once a baseline set of measures have been obtained and used to characterize HPCS development, the use of measurement can begin to support project management activities. In this application, measures taken from the project requirements or from in-process development can be compared to measures obtained from prior development efforts or used as input to the predictive models generated from these measures. These comparisons and model outputs can be used to help guide the new development. Possible forms of guidance include: the need for new or different kinds of resources, the appropriateness of the given workflow chosen for the goals of the project, and approaches to improving the quality of the system. Of course, the measures taken during every development can feed back into the characterization process to provide better understanding and modeling of HPCS development.

We propose to measure HPCS development in both qualitative and quantitative ways. Our measurement techniques will include structured interviews, time and motion studies, and automated measurement. Each of these techniques has differing kinds of strengths

and weaknesses. By employing all of these techniques in this research, we can ameliorate the weaknesses present in each form and improve the overall validity of the research.

3.1.1 Structured interviews

In structured interviews, a researcher talks directly with members of the development team, recording data using notebooks, audio tape, or video tape to learn more about the developer's view of the development process and its strengths and weaknesses. Structured interviews are useful for general characterization of a workflow, gaining insight into the kinds of quantitative measures that would be useful to collect, and collecting examples of process problems and solutions.

Some advantages of the structured interview process are that it is relatively inexpensive to carry out and does not require extensive access of the researcher into the development process. However, it suffers from the fact that the way a developer recalls a development situation could vary in significant ways from the reality of that development situation when it actually occurred. In addition, social or political pressures can influence the way a developer represents development obstacles or problems. Over time, a developer might simply forget or not perceive significant influences on development.

In many cases, however, we will be interested in data, observations, and lessons learned from projects that have already completed. For such projects, structured interviews are the only way to recover much of the information of interest (e.g., perspectives the time spent on different activities or the relative value of different system properties). Here the use of carefully structured interviews allows one to generalize over the data collected. This is the approach currently being used in a set of retrospective studies of high-performance computing applications being led by Doug Post of Los Alamos National Laboratory [Post 04]. Results from these project retrospectives will be used to help characterize application domains in terms of characteristic requirements and canonical work flows.

3.1.2 Time and motion studies

In time and motion studies, also known as "naturalistic observation," the observer spends time "shadowing" one or more members of the development team, recording the tasks to perform and the time intervals during which the tasks are performed. Developer logs can augment direct observations.

Time and motion studies have the advantages of supporting the development of fine-grained models of how developers spend their time, and surfacing issues in development process that might not otherwise be perceived by developers. The data that is collected is thus of generally higher quality and fidelity than that collected by structured interviews. The disadvantages of time and motion studies are access and cost. The organization and developers must be willing to allow a researcher to monitor their behavior over extended periods and measures must be taken to prevent that monitoring from interfering with the developer's behavior. In addition, the technique is very expensive, requiring the researcher to essentially work full-time at the institution doing data collection. Because of this, time and motion studies are usually restricted to just a few days of data collection. This means that only a small period of development during any given project can be monitored.

Several examples of software development time and motion studies appear in Perry, Staudenmayer, and Votta [Perry 96]. In that work, the authors summarized the results of three studies of developer time usage at Lucent Technologies.

The first experiment was an after the fact analysis of one software developer's personal diary. This diary recorded that developer's work-related activities over a 3-year period. This analysis helped the authors create initial hypotheses about time usage they tested in a later experiment. For example, analysis of the diary suggested that, much more than the authors expected, developers were blocked because important information was not immediately available to them.

The authors then conducted a second experiment to examine their newly generated hypotheses. To do this they created a data collection instrument accurate to about ½ hour of time. Next, they recruited a number of Lucent developers who agreed to fill out the data collection forms each day. From these data the authors made several observations. First, the subjects typically worked on two features simultaneously, so they could context switch when they became blocked. Second, even though context switching allows developers to keep working when blocked on one project, the cost is that individual projects take longer to finish. This was particularly important in the telephony industry where time-to-market pressures were severe.

Finally, the authors conducted a third study to compare self-reported time data with observed time data. The goal was to understand how accurate self-reported data is. This is important because it is much cheaper to collect data this way than to hire an external reporter. This study suggested that contrary to some people's beliefs, developers did not consciously misrepresent data. However, there were some systematic sources of bias. For example, some developers tended to lump short work interruptions (e.g., a colleague drops in for a 10 minute technical discussion) in with the activity they were conducting when interrupted.

Lessons learned from these studies are being applied to development and validation of our productivity benchmarks to ensure accurate representation of workflows and accurate data on units of work.

3.1.3 Automated measurement

A third form of measurement involves collection of data using the artifacts of development itself. For example, if development uses a source code control system, then the logs from this system can be analyzed to understand the patterns of developer interaction with the source files over time. The system itself can be generated at various points during its development to recover measures of its size or complexity.

Automated measurement has the advantages of collecting more objective measures of the process and products of development that are not filtered through the perceptions of developers. Automated measurement also has the advantage of being relatively low cost: researchers are not required to be on site, and developers do not have to deal with the potential short-term or long-term intrusion of a researcher in their daily activities. The disadvantage of automated measurement is its incompleteness relative to the other methods: many development activities cannot be reconstructed by analysis of the evidence left by tool usage.

An example of automated measurement is the work by Johnson on Hackystat [Johnson 03]. Hackystat is a system for automated metrics collection and analysis that provides custom "sensors" that are attached to developer tools, such as their editor, build tool, configuration management system, and so forth. The sensors unobtrusively monitor development and collect data about the process and products of development.

Some aspects of HPCS productivity measurement are similar to software productivity measurement in general. For example, automated metrics collection can provide an indication of how much time is spent actively modifying source code files. Automated tools can also monitor activities such as the invocation of test cases, and the sequence and nature of developer interactions with a configuration management repository. The tools can automatically gather data on the overall size of the system (expressed as numbers of non-comment lines of code, and/or number of methods, and/or number of compilation units), or complexity measures (such as measures of coupling between or cohesion within modules, and the characteristics of the inheritance hierarchy).

However, other aspects of HPCS productivity measurement will be required due to the specialized nature of this domain. For example, many workflows for HPCS application software development involve the initial development of a serial version of the system and measurement of its performance characteristics. Following the establishment of this baseline performance level, the next development stage involves implementing a parallelized version, typically using packages like MPI or OpenMP. Following this, experimental runs are performed to compute measures like Speedup, which indicates how much faster (or slower) the system executes as the number of processors allocated to the problem increases. Based upon the Speedup curve or other measures of parallel performance, the developer may decide to re-implement the parallelized version. By automatically monitoring the invocations of tools for compilation, execution, and performance profiling, it is possible to measure the time spent in each of these phases of HPC development.

3.2 Measuring Developmental Qualities ("ilities")

Obtaining any sort of fine-grained measures of productivity using our approach requires that we be able to provide relatively precise measures of progress, completion or other figures of merit for developmental properties like maintainability and portability. While such properties frequently appear in software requirements, the conventional wisdom is that they cannot be effectively measured. Where these properties are part of a system's acceptance criteria, industry standard practice is to judge them up or down by some form of peer review (e.g., Fagan inspections [Fagan 76]).

While our value equation certainly permits all-or-none completion criteria (i.e., value zero or one for one or more C_i in equation (2)), their use coarsens any measure of relative productivity. In many cases, users will be interested questions of degree: e.g., how much more portability do we get using development platform A compared to B. Further, if such properties account for a significant portion of a development's productive effort, we may lose the precision needed to make meaningful productivity comparisons. For this reason, it is important to develop effective metrics for developmental properties.

In the following, we illustrate one approach to providing just such a metric for the property commonly referred to as "maintainability." While we do not expect the specific approach described to generalize to other developmental properties, it does illustrate 1) that conventional wisdom is wrong, at least in this case and 2) there are some general principles that may be useful in developing metrics for similar kinds of properties.

3.2.1 Example: a Maintainability Metric

Our goal is to develop an architectural design for our software system that is "maintainable." By "maintainable," we mean that the system is relatively easy to change for expected types of changes[1]. In the general case, a single architectural design cannot be equally easy to change for all types of changes so the designer must choose which kinds of changes will be accommodated easily and which will not (e.g., cause dovetailing changes or "break" the architecture).

In short, before it makes sense to talk about (much less measure) maintainability, we must be precise about what we mean by the word. Any given architecture may be more maintainable than another relative to one set of changes but not to another. Thus, we must first answer the question: "Maintainable relative to what?" We answer this question by specifying a list of:

1) Anticipated *types* of changes. We list changes that potentially impact the architectural design. These should not be highly detailed; rather they should identify classes of change like: "It is expected that the Doppler radar will be replaced with a new model over the lifetime of the aircraft" or "It is likely that the pattern-matching algorithm will be replaced with a more efficient version."

2) Relative priority of each type of change. Specifying the priorities of different changes gives a measure of their relative value and allows the architect to make appropriate tradeoffs if necessary.

We now develop a metric that characterizes a design's maintainability relative to our list of expected changes. We characterize priorities as low, med, or high, and assign to them the respective weights 1, 3, and 9. Our architectural design strategy applies information hiding [Parnas 72]. That is, the designer seeks to 1) encapsulate each piece of information that is likely to change in exactly one module and 2) decompose the modules such that if two items are likely to change independently, then they will be encapsulated in different modules.

To the finished design, we apply a simple pass/fail completion criterion. If at most one module must be changed in response to each an anticipated change, then the design passes (receiving value 1); otherwise, the design fails (receiving value 0). The value of each change is then given by its priority times its completion value and the maintainability is given by the sum over the list of anticipated changes.

We can now calculate the maintainability metric associated with a particular architecture by "playing" the set of anticipated changes against the design. That is, each change is

[1] "Maintainability" has no precise definition and is used in the industry to cover virtually any kind of change to software following deployment. Our definition focuses on a one common aspect of what is usually meant by "maintainability."

treated as a scenario that a reviewer applies to the system. If the change can be made by making changes to only one module (i.e., the design is consistent with the information-hiding principle) then it passes for that change.

The result is a metric with granularity proportional to the number of anticipated changes that characterize "maintainability." Further, it is clear that, by its construction, the metric measures what the stakeholders mean by modifiability. Modifiability is defined to mean that anticipated changes are confined to one module. We then generate the change metric from the definitions of anticipated changes and produce the metric by simulating those changes.

There is, of course, considerable latitude in how we assign weights or values. For example:

1) We could construct a simple metric based on giving a value of "1" for each change confined to a single module and "0" otherwise—i.e., simply the number of changes that can be accommodated easily from a list of changes.
2) In practice, we have constructed metrics that include not only the priority but also the *likelihood* associated with each change. Lower weights can be assigned to changes of lower likelihood so the metric gives higher value to architectures that address the likely changes first.
3) We can express the maintainability metric a value in terms of the amount of rework or cost associated with making changes. Rather than pass/fail, we evaluate each change against an architecture by estimating the amount of rework (and cost) to implement the change. A standard estimation exercise for change requests. We then use the total rework (weighted as desired) as the relative metric of design maintainability. This actually gives the measure in a form directly translatable to productivity.

An objective of our empirical studies will be to develop comparable metrics and methods of measurement for other properties interest. These will be empirically validated and included in the purpose-based benchmarks.

4 Benchmark Development and Execution.

We are in the process of developing an initial set of benchmarks, metrics and tools to validate our conceptual approach to productivity measurement for HPCS. Our development approach is iterative:

1) Identify community of developers who will execute benchmarks.
2) Develop productivity measurement infrastructure appropriate for that community, e.g., define benchmarks, define workflows and corresponding functional- and non-functional requirements, create and install measurement instruments and analysis techniques,
3) Observe and measure developers as they execute the benchmark using the previously-defined infrastructure components, and
4) Analyze benchmark performance and evaluate and improve infrastructure.

We discuss these steps below.

Identifying the developer community. We have identified two initial developer communities who will participate in our studies. One is made up of computer science graduate students at the University of Maryland. The other is a group of professional software developers working remotely from Russia. We are working with these two communities because each presents different experimental design characteristics and cost/benefit tradeoffs for our research. In particular, these two communities represent different tradeoffs between internal validity, external validity, cost, and data quality. That is, the student community provides us with an *in vitro* experimental setting. Here we have substantial control over many aspects of observable behavior: the programming tasks, programming environment, outside influences, observation methods used, etc. In addition, the costs associated with observing students are relatively low, making prototyping more feasible.

As with most *in vitro* situations, however, control comes at the price of representativeness. The students are not usually professional developers (though some are), so the tasks must be restricted in their time length and complexity, etc. To gather data that is more representative of the complex software development workplace, we need access to an *in vivo* experimental setting. The professional developer community provides this. This setting allows for less control because we must be careful not to overly interfere with the developers. On the other hand, the data we do collect is likely to be more relevant to our overall goal of understanding HPCS productivity (stronger external validity).

Develop productivity measurement infrastructure. Here we have begun by identifying a general PBB. This problem involves writing software to compute "optimal" designs for a weight-bearing truss with certain material characteristics [Gustafson 04]. The original problem was specified with considerable detail. We are using this definition as given for the professional community, but are working to scale down the benchmark for the student community.

Each developer community will be following a different canonical workflow. At a high level, the professional community is following the Enterprise Workflow, while the students will be following the Lone Researcher Workflow (see Kepner overview). As part of this work, we are also defining the functional and non-functional requirements for the benchmark.

As discussed in Section 3.1, we are developing low-level data collection mechanisms (sensors) to capture developer activities unobtrusively. One novel way in which we will use this data is to develop low-level event traces (opening/closing files, running the compiler, etc) and match them to higher level actions defined by the canonical workflow. This will help us to better understand iteration, backtracking, and time usage within the overall process flows to help us better understand bottlenecks in the workflows.

Execute the purpose-based benchmark. Benchmark execution begins with an inoculation step. Participants are informed of the experimental procedures, explained about all data we will collect and why, informed of their right to withdraw from the study at any time, and are asked to provide informed consent. We then explain all the

functional and non-functional requirements required of the implemented benchmark. For the student community we will also conduct a tutorial on the engineering and mathematical concepts underlying mechanical trusses. Our goal here is to avoid any miscommunication, or apprehensions about the experiment. Initially, participants will also be asked to maintain some manual record to help us calibrate our automated techniques. Finally, participants will implement the benchmark in the context of the appropriate canonical workflow. Along with measuring overall productivity we also expect to: Identify and measure current bottlenecks in development process, to highlight differences between activity- and performance-based benchmarks, and to begin analyzing how different machine architectures address existing bottlenecks.

We will be executing our first pre-pilot study using CS graduate students in the spring semester 2004. We will simultaneously begin studying the professional developers.

Analyze benchmark performance, and evaluate and improve infrastructure. As we begin to get data from these studies, we will continuously monitor and improve our infrastructure.

5 Open Issues

Our approach to characterizing and measuring productivity is consistent with DARPA's programmatic goals of

- addressing both development-time and execution time productivity issues,
- providing benchmarks suitable for all the major classes of HPCS applications,
- measuring productivity in terms of the actual value created,
- addressing strategic development concerns outside of a single life cycle such as the value of transferable, portable, and reusable software products, and
- finding measures of productivity that users and vendors will agree on.

The (necessarily) open-ended framework allows us to define the output of development to include any properties necessary to characterize "value" for that development or class of developments. Indeed, any result to which we attach value and for which we can devise a measure can be included. Further, value expressions can be expanded or contracted as needed to meet new situations by adding new terms or removing existing ones.

However, the approach also will require substantial work to make it operational and effective.

1) The properties of interest and relative values for different application areas are not currently known. These will have to be established empirically through stakeholder interviews and review of existing codes or specifications.
2) For any property of interest, we will have to establish appropriate and effective metrics of completion (or comparable metrics of scale). For many properties such as portability, maintainability, etc. there is currently no agreed upon measure or even process for determining relative figures of merit. For each such property, we will either have to develop new approaches or fall back on imprecise gradation determined by review.

3) Our value equations [1] and [2] are only a first approximation. It is unclear yet to what extent we will need to address dependencies or how best to incorporate time into the model. In-depth examination of real development situations will be needed to make this clear.

4) Currently benchmarks do not address the range of properties we have discussed. Effective procedures will need to be developed for incorporating the properties of interest (e.g., as requirements), metrics, and measurement processes or tools into the benchmark specification. Likewise, we will need to develop directions on how to tailor the benchmarks and interpret the results of their execution.

6 Summary

Ensuring that next-generation HPC platforms significantly improve real productivity in terms of the science accomplished will require new approaches to characterizing, measuring, and predicting productivity. Current productivity metrics and benchmarks fall short in several ways. Traditional software metrics focus on measurable outputs but often bear little relationship to the actual value of what is produced. Common benchmarks tend to focus on machine performance, ignoring the growing bottlenecks associated with software development.

Our goal is to establish, apply, and validate an effective approach to assessing and predicting productivity that spans both development and execution time. Our objective is to provide these capabilities in a form that supports platform buyers in choosing the best system and platform developers in providing technology that addresses real productivity problems.

In this paper, we have described an empirical approach to understanding and addressing HPCS productivity. While it is clear that software development is increasingly a productivity issue in many HPC systems, few specifics are known. Before such problems can be addressed we need a better understanding of precisely what kinds of problems are occurring, where these problems occur in the process, how these differ from one process to the next, and how they vary from one type of HPC application domain to the next. We plan to collect such data through careful observation (e.g., interviews) of 1) real projects in common development domains and 2) experimental development efforts on canonical benchmark problems. Data from real development efforts will allow us to understand and catalog problems, requirements, and constraints characterizing different types of HPC applications. We can then characterize representative requirements and value propositions for different domains.

Carefully controlled experiments will help us better understand precisely where developers spend their time and how different platform features might increase the efficiency those activities. From this we expect to develop detailed canonical workflows representative of different development environments.

Detailed knowledge of problem characteristics, requirements, values, and workflows will be combined to develop tailored, productivity benchmarks for key HPCS domains. These benchmarks will provide not only a representative computation problem, but representative non-functional requirements as well. This includes domain-characteristic requirements for properties like portability, reliability, maintainability, etc. along with

appropriate measurement techniques. As a whole, each benchmark will exercise the entire development process across a value space appropriate to the domain and provide metrics and tools for measuring productivity throughout the development cycle.

7 Acknowledgements

We thank Mr. Dale Shires of the U.S. Army Research Laboratory High Performance Computing Division for permission to use portions of the VPATC [VPATC 03] specification. We also thank Dr. Doug Post of the Los Alamos National Laboratory and Mr. Andy Mark of the High Performance Computing Modernization Program Office for their help in obtaining the specification. Thanks also to Bill Walster, Dolores Shaffer, and our anonymous reviewers for their careful reviews and thoughtful comments.

8 References

[Albrecht 83] Albrecht, A. and Gaffney, J., "Software function, source lines of code, and development effort prediction: a software science validation," *IEEE Transactions on Software Engineering*, vol. 9, no. 6, pp. 639-648, 1983.

[Boehm 80] Boehm, B. and Wolverton, R. W.,"Software Cost Modelling: Some Lessons Learned," *Journal of Systems and Software*, vol. 1, no. 3, pp. 195-201, 1980.

[Boehm 81] Boehm, B., *Software Engineering Economics*, Prentice Hall, 1981.

[Boehm 95] Boehm, B., Clark, B., Horowitz, E., Westland, C., Madachy, R., and Selby, R., "Cost models for future software life cycle processes: COCOMO 2.0," *Annals of Software Engineering*, vol. 1, pp. 57-94, 1995.

[Brooks 95] Brooks, F. P., *The Mythical Man-Month: Essays on Software Engineering, Anniversary Edition*. Addison-Wesley, New York, ISBN 0201835959, 1995.

[Jones 86] Jones, C., *Programming Productivity*, McGraw-Hill, New York, 1986.

[Fagan 76] Fagan, M., "Design and Code Inspections to Reduce Errors in Program Development," *IBM Systems Journal*, vol. 15, no. 3, pp. 182-211, 1976.

[Johnson 03] Johnson, P., Kou, H., Agustin, J., Chan, C., Moore, C. A., Miglani, J., Zhen, S., and Doane, W. E., "Beyond the Personal Software Process: Metrics collection and analysis for the differently disciplined," *Proceedings of the 2003 International Conference on Software Engineering*, Portland, Oregon, May, 2003.

[Kepner 04] Kepner, J., "HPC Productivity: an Overarching View," *International Journal of High Performance Computing and Applications: Special Issue on HPC Productivity* (ed. Kepner), vol. 18, no. 4, Winter 2004

[Kitchenham 92] Kitchenham, B. A, “Empirical studies of assumptions that underlie software cost estimation,” *Information and Software Technology*, vol. 34, no. 4, pp. 211-218, 1992.

[Gause 89] Gause, D. and Weinberg, G., *Exploring Requirements: Quality Before Design*, Dorset House, 1989.

[Parnas 72] Parnas, D. L., “On the criteria to be used in decomposing systems into modules,” *Communications of the ACM*, vol. 15, no. 12, pp. 1053-1058, 1972.

[Perry 96] Perry, D., Staudenmayer, N., and Votta, L., "Understanding and Improving Time Usage in Software Development," in *Trends in Software: Software Process*, Wolf and Fuggetta, editors, John Wiley & Sons, 1996

[Post 04] Post, D. and Kendall, R., “Software Project management and Quality Engineering Practices for Complex, Coupled, Multi-Physics, Massively Parallel Computation Simulations: Lessons Learned from ASCI,” *International Journal of High Performance Computing and Applications: Special Issue on HPC Productivity* (ed. Kepner), vol. 18, no. 4, Winter 2004.

[Porter 95] Porter, A., Votta, L., and Basil,i V., “Comparing detection methods for software requirement inspections: A replicated experiment,” *IEEE Transactions on Software Engineering*, vol. 21, no. 6, pp. 563-575, June 1995.

[Snir 04] Snir, M. and Bade, D., “A framework for measuring supercomputer productivity,” *International Journal of High Performance Computing and Applications: Special Issue on HPC Productivity* (ed. Kepner), vol. 18, no. 4, Winter 2004.

[VPATC 03] Software Development Plan for Virtual Prototyping and Accelerated Testing of DoD Composite Material Combat Systems (VPATC), SOS-3, April 2003.

[Weinberg 74], Weinberg, G. and Schulman, E., “Goals and performance in computer programming,” *Human Factors*, vol. 16, no. 1, pp. 70-77, 1974.