Defining and Measuring the Productivity of Programming Languages[†]

Ken Kennedy, Charles Koelbel
Computer Science Department, Rice University
Robert Schreiber
Laboratories, Hewlett Packard Company

The overall objective of programming support systems is to make it possible to produce software faster with the same workforce, with no degradation, and possibly an improvement, in software quality. Generally, there are two ways to approach this goal. First, we can increase the effectiveness of individual application developers by providing programming languages and tools that enhance programming productivity. Second, we can broaden the community of application developers by making programming more accessible. As it happens, the use of higher-level languages and programming interfaces supports both these strategies: by incorporating a higher level of abstraction, such languages make application development both easier and faster. (For the purposes of this paper, we will define "programming language" to encompass the entire toolset—language, compiler, debugger, tuning tools—associated with the language.)

We must, however, ensure that these advantages do not come at the cost of performance. Programs written in a high-level language and intended to solve large problems on highly parallel machines must not be egregiously less efficient than the same applications written in a lower-level language. If they are, then the language is unlikely to be accepted. Because this has been a traditional stumbling block for high-level languages, our productivity analysis must incorporate metrics of both programming effort and performance. Furthermore these metrics must be linked so that the trade-off between language power and program efficiency can be evaluated properly.

Similarly, if high-level languages are to be accepted, programs written in them cannot exhibit more faults, consume more memory, or be less portable than if written in low-level competitors. Fortunately, these have not been troublesome issues in the past, so we feel justified in not addressing them head-on in this paper, although we do feel that such factors should be investigated in the future.

Thus, for any given development task, each programming language must be evaluated with respect to at least two criteria: the *time and effort* required to write, debug, and tune the code, and the *performance* of the code that results. The goal of this paper is to define these two evaluation metrics clearly and unambiguously and to propose methods by which to measure them.

Primary Metric: Time to Solution

By increasing productivity, we aim, in the end, to reduce total time to solution of a problem P, which we denote $T(P)$. That is, we want to minimize

(1) $$T(P) = I(P) + rE(P)$$

where P is the problem of interest, $I(P)$ is the implementation time for a program to solve P, $E(P)$ is the average execution time per run of the program, and r is a problem-specific weighting factor that reflects the relative importance of minimizing execution time versus implementation time. Thus, r will be larger for programs that are to be run many times without any change.

If it is more appropriate to focus on the cost of generating the solution than on the time to solution, we arrive at essentially the same model:

$$C(P) = D(P) + rM(P)$$

where $D(P)$ is the cost of developing the application code and $M(P)$ is the average cost of machine time for one run of the application. Because of the isomorphism between the time and cost models, most of the strategies we propose can be applied to both problems. For our present purpose—developing a strategy for defining and assessing the productivity of a programming interface to a machine—program development (and tuning) time and production run time are the first-order, measurable impacts of the choices we make; therefore, we focus on time rather than monetary cost.

The formulation in Equation (1) assumes that "implementation" and "execution" are activities that do not overlap. This is clearly an oversimplification, but to our mind an acceptable one. However, the formula leaves us with the vexing problem of choosing for a particular application the appropriate value for r, which estimates how many times that application will be executed. If the application is only to be run once, then r can be 1, but most important applications will be run many times with no change between runs, except for data set changes and minor source tweaks. It is therefore critical to ensure that r is not underestimated. How does one know how many times a program will be used over the course of its lifetime? We may need to rely on opinion to get an answer. Indeed, all performance programmers must be implicitly making and using an estimate of this parameter; for in any program tuning effort, further work is stopped at a point of diminishing returns, where the added development time is not compensated for by an anticipated reduction in execution time as weighted by an estimate, perhaps only a tacit one, of r. Thus, while r cannot be known (except in rare cases after a program is finally retired), we think it can be usefully approximated. We discuss the use of expert opinion in estimating parameters of the model in the section "Measuring Programmer Performance and Power", below.

Note that formulation (1) does not include the notion of compile time. In current languages, compile time is generally small relative to the running time of the object code. However, more advanced languages may require whole-program compilation, which

could elevate compilation time to a significant factor in return for lowered execution time. At the other end of this spectrum, interpreted languages remove the compile time penalty, albeit at execution time cost; for applications that exhibit low r or low $E(P)$, the resulting gain in user productivity surely helps to explain their popularity (Cleve Moler estimates that Matlab has approximately one million users). This raises the question of whether compile time should be incorporated into the model explicitly. If so, should it be an independent term, of should it be bundled in some way with implementation time or execution time. Our inclination is include it as a third, independent term. Thus, we have the more detailed time-to-solution model

$$(2) \qquad\qquad T(P) = I(P) + r_1 C(P) + r_2 E(P)$$

where $C(P)$ is the average compilation time, r_1 is the number of compilations and r_2 is the number of runs. Where compile time is a first-order issue, this model is preferable, but to keep things simpler we drop the compile-time term and user the simple model (1) in the remainder of the paper.

Relative Efficiency and Relative Power of a Programming Language

The implementation and execution times $I(P)$ and $E(P)$, which we have explicitly shown to be dependent on the problem being solved or program being written, clearly depend on many other things. Among these are the algorithm used, the data structures, the machine, its architecture and its degree of parallelism, the programming language, and the programming team and its experience with problem, machine, and language.

Our central thesis, however, is that within reasonably bounded and interesting situations, such as the mix of problems solved at a given supercomputer site, we can meaningfully define and measure the relative efficiency and power of two proposed programming languages. Furthermore, we suspect strongly that in comparing different programming languages, the relative power and efficiency will not vary widely between sites, but rather are genuine attributes of the language itself. We therefore propose two derived measures, *relative power*, denoted ρ, and *efficiency*, denoted ε, as productivity metrics. These relative measures can be used to compare programming systems.

It will be useful, in any given context (a given problem, a given machine) to define some programming language and its toolset as the basis for comparison. Thus, let P_0 be a version of a given program written by a professional in a standard programming language, which we take as a reference point. In some contexts, studies on uniprocessors for example, this base case might be a sequential Fortran program. On a small SMP it might be C with the OpenMP extensions, and on a large cluster it could be Fortran with Message Passing Interface (MPI). We compare the implementation and execution time of P_0 with those of P_L, the equivalent program in a new programming language. The relative power ρ_L of the language, measuring its ease of use, is the ratio of implementation times, while the efficiency ε_L, measuring performance of the language, is the ratio of execution times. That is,

$$\rho_L = \frac{I(P_0)}{I(P_L)}$$

(3)

$$\varepsilon_L = \frac{E(P_0)}{E(P_L)}$$

Generally, $\rho_L > 1$ and $\varepsilon_L < 1$ for high-level languages, reflecting the tradeoff between abstraction and performance. Both ρ and ε depend on the application; however, we believe that ρ and ε are relatively constant for a particular language: that is, they do not vary by orders of magnitude, at least within its chosen domain.

Assigning values to ρ and ε requires developing two programs in a controlled environment where $I(P)$ and $E(P)$ can be measured. This is practical for relatively small benchmark problems. (Of course, running many small experiments can be very expensive, too; it is therefore important that the set of benchmarks be relatively compact.). For large-scale projects, it is unrealistic to expect that two or more different languages would be used in independent, parallel efforts. If one team of programmers does both implementations, the comparison is rendered invalid by the experience (concerning the problem) gained in the first effort.

Relating this back to our goal of reducing time to solution, we find that

$$T(P_L) = I(P_L) + rE(P_L)$$

$$= I(P_0) \cdot \frac{I(P_L)}{I(P_0)} + rE(P_0) \cdot \frac{E(P_L)}{E(P_0)}$$

$$= \frac{1}{\rho_L} I(P_0) + \frac{1}{\varepsilon_L} rE(P_0)$$

In other words, once we have good estimates for ρ_L and ε_L, it is relatively easy to estimate the time to implement and run an application in a new language L if we know the implementation and execution times for the base language. With given values of r, $I(P_0)$ and $E(P_0)$, there may be several tuples (ρ, ε) that minimize $T(P_L)$. It is therefore important to consider both metrics in choosing the programming system. A convenient way to present these is as a graph in which relative power is displayed on the y-axis and efficiency on the x-axis, as shown in Figure 1. Contours on this graph can show productivity tradeoffs. In particular, a contour can be plotted to show (ρ, ε) pairs that lead to identical $T(P_L)$.
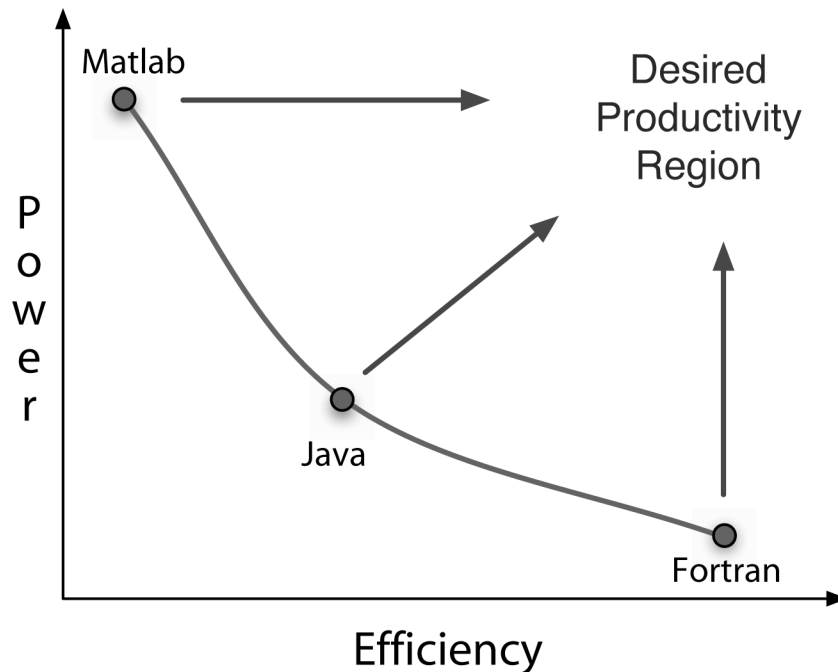
Figure 1. Power-efficiency graph.

The graph in Figure 1 illuminates an important issue: how one can improve productivity by improving programming languages and their implementation. Generally, this can be done in two ways (or combinations thereof). First, one can take existing high-performance languages and improve their power, possibly by adding advanced features, without sacrificing their performance. Thus, Fortran could be improved by adding powerful new language primitives; this was a goal of the Fortran 90, 95, and 2000 efforts. Unfortunately, the performance of applications written in these new generations of Fortran has been compromised by the inefficient object code generated by immature compiler technology, thus reducing the expected productivity gains. Second, one can enhance productivity by improving the performance of very high-level languages such as Matlab. This is the goal of a number of efforts to provide full or partial compilation facilities for Matlab [DeRose, Chauhan].

We propose to use graphs like the one in Figure 1, rather than a simple scalar measurement, as the main way of displaying productivity to users. Note that this lets the end user determine whether a language is powerful enough to compensate for the incurred level of inefficiency. It allows us to reason about productivity goals in terms of the relative power-efficiency plane.

It will be quite instructive to plot (ρ, ε) pairs for a collection of data points, each generated by a different application, to verify our assertion that they will cluster, with one cluster

for each distinct, interesting language and toolset. If the clustered structure is not so simple, we will have learned something interesting about our programming languages.

Although we believe this two-dimensional representation to be the best way to display language productivity, it may nevertheless be useful to distill the measurements of performance and effort across tasks into a single quantity that characterizes the productivity achieved by the use of a given language, recognizing that any reduction from the multidimensional measured data to a single scalar loses information. This suggests that a plausible metric for the productivity of a language is some measure that incorporates its relative power and the relative performance of its object code. In order to combine these two dimensionless quantities in a meaningful way, we need an exchange rate. Consider that the ratio of times to solution is

$$\frac{T(P_0)}{T(P_L)} = \frac{I(P_0) + rE(P_0)}{I(P_L) + rE(P_L)}$$

$$= \frac{\rho_L I(P_L) + \varepsilon_L rE(P_L)}{I(P_L) + rE(P_L)}$$

$$= \frac{\rho_L + \varepsilon_L X}{1 + X}$$

where $X = r\, E(P_L) / I(P_L)$. Thus, the problem-dependent parameter X allows us to combine the relative power and the efficiency of a programming language into a relative productivity metric:

(4) $$productivity = \frac{\rho + \varepsilon X}{1 + X}$$

If, as we assert above, the relative power and the efficiency of a language are largely problem independent, then it follows that the effect of the problem on overall *productivity*, defined as the relative time to solution, is captured by the problem-dependent parameter X. For the base language L_0, $\rho = \varepsilon = 1$; hence *productivity* for the base language is fixed at 1. From equation (4), it is easy to see why, for long-running applications, programmers might be willing to sacrifice language power to achieve much higher efficiencies through parallelism. For example, if it took five times longer to reprogram a Fortran application using MPI to achieve a speedup of 50, the overall productivity of Fortran plus MPI, relative to single-processor processor Fortran, would be at least 25, assuming that the ratio X of total running time to implementation time remained greater than 1.

For languages of higher power than the base, where we expect that $\rho > 1$ and $\varepsilon < 1$, *productivity,* as defined by equation (4), declines with increasing values of X from a high of ρ at $X=0$ to asymptotically approach a lower bound of ε as X grows. This behavior is illustrated by the curve in Figure 2.
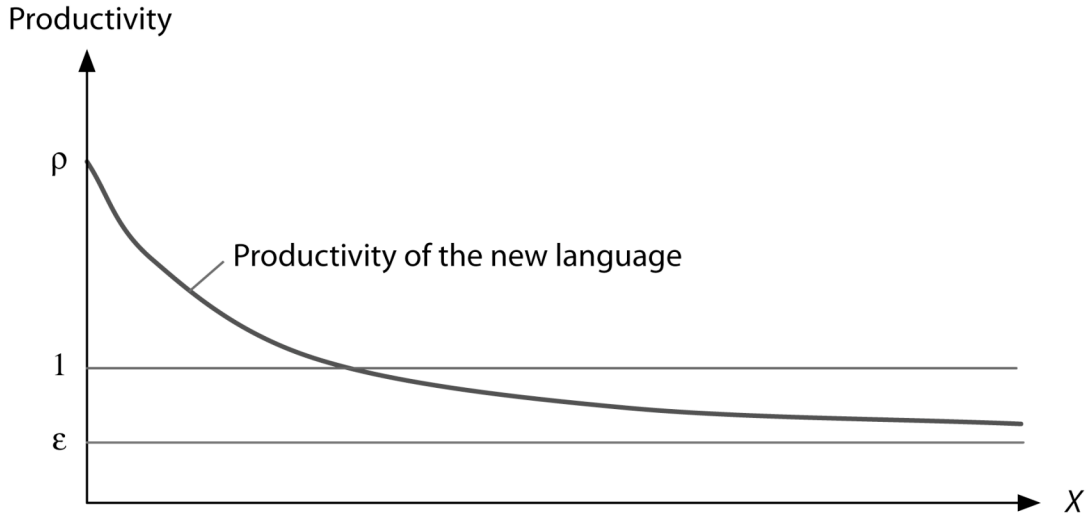
Figure 2. Productivity as a function of running time over implementation time.

Figure 2 also shows the importance of achieving high efficiency in a high-productivity language. For example, if some new language has a power that is two times greater than the base language, but only half the efficiency, *productivity* is 1.25 when *X*=1 and it equals that of the base language when *X*=2. If, on the other hand, the efficiency of the new language is 75 percent of that of the base language, the crossover point occurs at *X*=4, illustrating the value of rather modest improvements in efficiency. Generalizing, if we define X_1 as the value of *X* for which *productivity* of the new language is the same as for the base language (*i.e., productivity* = 1), then from equation (4) we get

$$X_1 = \frac{\rho - 1}{1 - \varepsilon}.$$

These discussions lead us to consider whether there is some lower bound on acceptable language performance for long-running applications. As an illustration, consider Matlab. In the high-performance computing community today, the power of Matlab and its implementation is not enough to compensate for the low performance delivered by the Matlab interpreter on large-scale applications. Thus Matlab remains a prototyping language and "serious" applications are usually rewritten in a lower level programming language. This implies a second effect, not directly captured by the metrics ρ and ε alone: there is a minimum level of acceptable performance, and a programming language that fails to deliver at this minimum level cannot be useful for "production" applications, no matter how greatly it reduces programming time and effort. This effect is reflected in the model above by *productivity* dropping below 1 for a relatively small value of *X*.

Measuring Application Performance and Efficiency

There is a well-developed literature for measuring application performance in parallel systems, which we will not try to summarize here. Suffice it to say that experience has

shown that standard sets of benchmarks provide a fair means of comparing systems (including hardware, software, and languages).

We are examining the relationship between program development time and execution time, as they are influenced by choice of language. In so doing, it is important to consider the full power of a new language. It may be possible to write Fortran-like code in Java, but that would tell us little about the power or efficiency of Java. Thus, when comparing two programming models on the same application benchmark, it is essential that the application be coded in a natural style for each of the programming models. The goal is not to show that you *can* write a program of comparable performance in a higher-level programming system, but rather to measure the performance cost paid when you write that application in the most natural style. Thus it is best that the different versions of the applications be written by developers with no knowledge of how the optimizing compilers work for each model.

It is also vital that the set of benchmarks be representative of the actual applications run on the system, lest the system developers optimize inappropriately. The apocryphal story of an OS team achieving a 50% performance gain in the system's idle wait loop comes to mind. Care must be taken to specify the rules (as was done in the NAS parallel benchmarks [NAS]) so as to eliminate the temptation to indulge in one of Bailey's twelve abuses [Bailey].

Given accurate measures of execution time, it is simple to calculate ε from its definition:

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

Measuring Programmer Performance and Relative Power

This leaves us with the problem of measuring relative language power, which is much more challenging. While it is often possible to agree that some languages are more powerful (or easier to program in) than others, it is extremely difficult to measure in any precise way the degree of difference. What we would really like to measure is development time. In other words, we need a metric that fairly compares the time taken to produce the same application by programmers starting out at the same time in the two different languages.

In their paper in the present issue, Faulk *et al*. describe several standard measurement techniques, their advantages, and their drawbacks [Faulk]. We consider these techniques here for the purpose of measuring program development time. In the terminology of Faulk, they are "time and motion studies," which is the direct measurement of program development time, the "structured interview" technique, in which the opinions of experienced developers constitute the primary data, and "automated measurement," in which the program development tools generate useful effort-related statistics as a byproduct of their use by programmers.

Automating the tools would offer a straightforward way to measure compile time, a component of the detailed model (2). Statistics like the number of compiles per day might prove to be informative as well, as might the nature of the changes to program text between compiles. Tools can provide counts of lines, statements, "function points" and other direct code metrics. However, the often-used "source lines of code" metric does not appear to us to be generally reliable as a direct predictor of program development time, especially when programs in different languages are to be compared. Development time encompasses coding time, debugging time and performance tuning time; the average coding, debugging, and tuning time per line of code may vary from one language to another. For example, it is generally conceded that a Java program takes less time to develop than a C++ program of roughly the same length because Java is strongly typed and garbage collected, so there are fewer opportunities to introduce errors. Thus, some models (see the paper by Post and Kendall in the present issue [Post]) apply a language-specific multiplicative factor to the lines-of-code measure. Further study is clearly needed to decide on the proper role of automated measurement as a way to assess and understand development effort and time, and the effect of programming tools on productivity. A disadvantage, clearly, is that these tools provide their estimates as the program is developed, not before. Two parallel efforts, in separate languages, would be needed to measure their relative power.

To measure implementation time directly, we would set up experiments in which two different sets of programmers implement the same application in two different programming languages. Then the power of the higher-level language would be the ratio of implementation time in the lower-level language to implementation time of the same program in the higher-level language, based on the previous definition:

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

Unfortunately, measurements of this sort, which seem tractable in principle, usually turn out to be invalid because it is difficult to factor out individual ability, particularly in a task like programming where some remarkable individuals can be integer factors more productive than others and where the resources devoted to the measurement task make a sufficiently large sample group impossible. Moreover, if L is a newly proposed language, there will not generally be a pool of talented, experienced L programmers. A comparison of novice L programmers against novice Fortran programmers would not be a valuable indicator of anything.

Because of these difficulties, we believe that expert opinions, solicited through carefully crafted questions before, after, or during a project, can generate useful estimates of relevant information, including program development effort. In other words, we will need to incorporate the views of the end users in evaluating the productivity scores of different languages, including their best guesses as to the relative power compared to a base language. As an example, consider Matlab. Most scientists and engineers now believe that programming in Matlab is far more productive than programming in, say, Fortran. So how much more productive is Matlab than Fortran? Programmers may have an opinion

about this, and we can measure their opinions. In other words, it may be possible, through structured interviews to have expert programmers estimate the time to completion of an implementation of each of a set of benchmark problems in both the new programming language L and the base language L_0. These opinions can then be aggregated into cumulative distribution functions that can be used to estimate $I(P_L)$ and $I(P_0)$. Strategies for obtaining and integrating the opinions of a group of experts have been used in the well-known "Delphi" method [Dalkey] and in computer-human interface evaluations for many years. Recent enhancements of methods for the aggregation of expert opinions offer promise for improving the predictive power of this approach [Chen].

Other Issues

So far, we have assumed that each language has a single value of ρ and a single value of ε for all relevant computations. This is seldom (if ever) precisely the case. We can compensate by measuring power and efficiency for a standard collection of benchmarks or a collection particularly suitable for a given customer or computer purchase.

If a single power or efficiency metric is desired, then an overall value can be computed from the individual benchmarks. Let $\{P_0^1, P_0^2, ..., P_0^n\}$ be the set of benchmark programs written in the base language and let $\{P_L^1, P_L^2, ..., P_L^n\}$ be the programs written in the higher-level language. Then the power and efficiency for that collection of benchmarks is:

$$\varepsilon_L = \frac{\sum_{j=1}^{n} E(P_0^j)}{\sum_{j=1}^{n} E(P_L^j)}$$

$$\rho_L = \frac{\sum_{j=1}^{n} I(P_0^j)}{\sum_{j=1}^{n} I(P_L^j)}$$

If desired, the times for the benchmarks could be weighted to reflect their importance, or to normalize for expected execution times different from the benchmarks. Alternatively we may use the ratio of the sum over all programs of time-to-solution. But, as stated above, we think that the best use of a collection like this is to plot all the data in the power/efficiency space and observe the clusters that emerge.

The ratio of sums is preferable to an arithmetic mean of the individual ratios because the arithmetic mean would emphasize outliers. For example, if language 0 is *really* great on one benchmark it creates an enormous ratio for that benchmark which gets averaged in. The ratio of weighted sums is more defensible as the ratio of runtimes on a hypothetical composite workload consisting of a weighted mix of the given workloads. Another outlier-tolerant approach could be to use the median of the individual ratios.

We have used time as the primary measurable quantity for both implementation and execution. Depending on the context, it may make sense to use other measures. One that is rather natural in large procurements is cost. On the implementation side ($I(P)$ and ρ),

this would represent the cost of programmer time, and possibly cost of ownership of development machines. On the execution side ($E(P)$ and ε), this would represent the cost of the machine time, which could take all costs of ownership (purchase, staffing, electric bill) into account. One still measures largely the same primary data (runtimes, program development hours) and then applies a time-to-cost conversion. Our definitions or relative power, efficiency, and productivity can still be used.

The value of the computed result is another important facet of the productivity problem. If it is a constant for the given problem P, independent of all other factors, then little changes. If, as is often true, the computation has some time value, then using time-to-solution as the appropriate metric still seems right. Measuring the value of computations directly is hard, but if the need arises, intelligent aggregation of market-based estimates, as advocated by Chen, Fine, and Huberman, [Chen] will be an interesting approach.

Other issues are not so easily incorporated into the analysis above. For example, if a code will be modified many times over its lifetime, the assumption of separate development and execution passes needs re-examination. We would want a way to quantify maintainability, modifiability, code reuse, and so forth. Defining and measuring correctness, security, maintainability, reusability, or other less qualitative properties of codes is difficult: Faulk *et al*. report that these virtues are generally viewed as impossible to accurately measure. One approach would be to allow subjective ratings for these aspects of the process, and report significant differences in ratings between languages. Faulk proposes a way to do this for maintainability, as an example of this approach. Given our language-centric point of view, an interesting question is whether, and to what extent, the programming language (as opposed to the programming style adopted) influences the reusability of HPC code developed in it, and whether there is a reusability-efficiency tradeoff like the relative power-efficiency tradeoff discussed in this paper.

Summary

We propose the use of two dimensionless ratios, relative power ρ and relative efficiency ε, to measure the productivity of programming interfaces. Determining values of these metrics for a new language may require writing a fixed class of benchmarks in the new language and measuring the implementation effort and running time incurred. These measures will be compared with the corresponding measures for a base language such as C or Fortran. (We note that the measurement of implementation effort, and of performance, may involve subjective measure such as the opinions of developers.) Rather than combine these metrics into a single number representing a universal productivity, we propose that they be represented graphically in at least two dimensions so that the trade-offs between abstraction and performance are clearly depicted. On the other hand, we introduce a single problem-dependent parameter that allows us to reason about the relative productivity of two languages for a given problem.

References

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

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

[Chauhan] A. Chauhan, C. McCosh, K. Kennedy, and R. Hanson. Automatic type-driven library generation for telescoping languages. In *Proceedings of the ACM International Conference for High Performance Computing and Communications (SC2003)*, Phoenix, Arizona, November 15–21, 2003.

[Chen] Kay-Yut Chen, Leslie R. Fine and Bernardo A. Huberman. Predicting the Future. *Information Systems Frontiers* 5:1, 47 – 61, 2003.

[Dalkey] Norman C. Dalkey. Analyses from a group opinion study. *Futures* 2:12, 541– 551, 1969.

[DeRose] L. DeRose and D. Padua. Techniques for the translation of MATLAB programs into Fortran 90. ACM Transactions on Programming Languages and Systems, 21:2, 286–323, Mar. 1999.

[Faulk] Stuart Faulk, Philip Johnson, Adam Porter, Walter Tichy, and Lawrence Votta. Measuring HPC Productivity. This issue.

[Post] D. E. Post and R. P. Kendall. Software Project Management and Quality Engineering Practices for Complex, Coupled Multi-Physics, Massively Parallel Computational Simulations: Lessons Learned from ASCI. This issue.