

# Application of a Development Time Productivity Metric to Parallel Software Development

Andrew Funk<sup>1</sup>  
afunk@ll.mit.edu

Victor Basili<sup>2</sup>  
basili@cs.umd.edu

Lorin Hochstein<sup>2</sup>  
lorin@cs.umd.edu

Jeremy Kepner<sup>1</sup>  
kepner@ll.mit.edu

## ABSTRACT

Evaluation of High Performance Computing (HPC) systems should take into account software development time productivity in addition to hardware performance, cost, and other factors. We propose a new metric for HPC software development time productivity, defined as the ratio of relative runtime performance to relative programmer effort. This formula has been used to analyze several HPC benchmark codes and classroom programming assignments. The results of this analysis show consistent trends for various programming models. This method enables a high-level evaluation of development time productivity for a given code implementation, which is essential to the task of estimating cost associated with HPC software development.

## 1. INTRODUCTION

One of the main goals of the DARPA High Productivity Computing Systems (HPCS) program [1] is to develop a method of quantifying and measuring the productivity of High Performance Computing (HPC) systems. At a high level, HPCS Productivity,  $\Psi$ , has been defined as utility over cost:

$$\Psi = \frac{U(T)}{C_S + C_O + C_M} \quad [2],$$

where utility,  $U(T)$ , is a function of time. Generally speaking, the longer the time to solution, the lower the utility of that solution will be. The denominator of the formula is a sum of costs – software ( $C_S$ ), operator ( $C_O$ ), and machine ( $C_M$ ). A higher utility and lower overall cost lead to a greater productivity for a given system.

---

This work is sponsored by Defense Advanced Research Projects Administration, under Air Force Contract FA8721-05-C-0002. Opinions, interpretations, conclusions and recommendations are those of the author and are not necessarily endorsed by the United States Government.

Efforts to quantify time to solution in the HPC community have traditionally focused on measuring execution time and developing benchmarks and metrics to evaluate computational throughput. However, few metrics exist for evaluating development time, which is increasingly being recognized as a significant component of overall time to solution.

Herein we define a new metric which we call development time productivity. In this case the utility is measured by how much faster a parallel code can find a solution, relative to a baseline serial code. The cost is measured by how much effort the programmer must put in to writing the parallel code, relative to the serial code. In other words,

$$\text{Development Time Productivity} = \frac{\text{Speedup}}{\text{Relative Effort}},$$

where

$$\text{Speedup} = \frac{\text{Serial Runtime}}{\text{Parallel Runtime}},$$

and

$$\text{Relative Effort} = \frac{\text{Parallel Effort}}{\text{Serial Effort}}.$$

Effort may be measured in various ways. The most direct way is to measure the actual time spent programming both the serial and parallel code. This is perhaps the most accurate measure, but it can also be problematic to obtain accurate time logs, and often this data is simply not available. In this case other software metrics, such as Source Lines of Code (SLOC), may be used instead.

Some studies have shown SLOC to correlate well with developer effort [3], though this is still open to debate. In any case, we are working in relative, not absolute terms. For example, if parallel code A requires 2x the SLOC of a baseline serial code, and parallel code B requires only 1.5x the SLOC of the baseline serial code, then it is reasonable to assume that code A required a larger amount of effort than code B to develop.

---

<sup>1</sup> MIT Lincoln Laboratory  
244 Wood Street  
Lexington, MA 02420

<sup>2</sup> University of Maryland  
Computer Science Department  
A.V. Williams Building, Room 4111  
College Park, MD 20742

We have applied this development time productivity formula to performance and effort data collected from two HPC benchmark suites, and from a series of graduate student parallel programming assignments. Section 2 describes in detail how the data were collected and analyzed in each case. The results of this analysis are presented in Section 3, and discussed further in Section 4.

## 2. ANALYSIS

### 2.1. NAS PARALLEL BENCHMARKS

The NAS Parallel Benchmark (NPB) [4] suite consists of five kernel benchmarks and three pseudo-applications from the field of computational fluid dynamics. The NPB presents an excellent resource for this study, in that it provides multiple language implementations of each benchmark. The exact codes used were the C/Fortran (serial, OpenMP), and Java implementations from NPB-3.0, and the C/Fortran (MPI) implementations from NPB-2.4. In addition, a parallel ZPL [5] implementation and two serial Matlab implementations were also included in the study.

These codes were all run on an IBM p655 multiprocessor computer using the Class A problem size. The parallel codes were run using four processors. The runtimes used were those reported by the benchmark codes. As discussed in the previous section, the speedup for each parallel code (and the serial Matlab codes) was calculated by dividing the baseline serial C/Fortran runtime by the parallel runtime.

The SLOC for each benchmark code was counted automatically using the SLOCcount tool [6]. For each implementation, the relative SLOC was calculated by dividing the parallel SLOC by the baseline serial C/Fortran SLOC.

For each benchmark implementation, a development time productivity value was calculated by dividing the speedup by the relative SLOC. The results of this analysis are presented in Section 3.1.

### 2.2. HPC CHALLENGE

The HPC Challenge suite [7] consists of several activity-based benchmarks designed to test various aspects of a computing platform. The four benchmarks used in this study were FFT (v0.6a), High Performance Linpack (HPL, v0.6a), RandomAccess (v0.5b), and Stream (v0.6a).

These codes were run on the Lincoln Laboratory Grid (LLGrid), a cluster composed of 80 dual-processor nodes connected by Gigabit Ethernet [8]. The parallel codes were run using 64 of these dual-processor nodes, for a total of 128 CPUs. The speedup for each parallel code was determined by dividing the runtime for a baseline serial C/Fortran code by the runtime for the parallel code (the serial Matlab code was treated the same as the parallel codes for purposes of comparison).

The relative SLOC (again counted using SLOCcount) was calculated by dividing the SLOC for each parallel code by the SLOC for a baseline serial C/Fortran code.

The development time productivity for each benchmark implementation was determined by dividing the speedup by the relative SLOC. The results of this analysis are presented in Section 3.2.

## 2.3. CLASSROOM ASSIGNMENTS

A series of classroom experiments were conducted for the HPCS program, in which students from several different classes were asked to produce parallel programming solutions to a variety of textbook problems (see Table 1). In most cases the students first created a serial program to solve the problem, and this was used as the baseline for comparison with their parallel solution. The students used C, Fortran, and Matlab for their serial codes, and created parallel versions using MPI, OpenMP, and Matlab\*P (aka StarP, a parallel extension to Matlab) [9].

Table 1. Classroom Assignments

Class	Problem	Programming Task	Students reporting
P0A1, P1A1	Game of Life	Create serial and parallel versions using C and MPI	16, 11
P0A2	Weather Sim	Add OpenMP directives to existing serial Fortran code	17
P2A1	Buffon-Laplace Needle	Create serial versions using C and Matlab, and parallel versions using MPI, OpenMP, and StarP	11
P2A2	Grid of Resistors	Create serial versions using C and Matlab, and parallel versions using MPI, OpenMP, and StarP	11
P3A1	Buffon-Laplace Needle	Create serial versions using C and Matlab, and parallel versions using MPI, OpenMP, and StarP	17
P3A2	Parallel Sorting	Create serial and parallel versions using C and StarP	13
P3A3	Game of Life	Create serial and parallel versions using C, MPI, OpenMP	8

The students ran their programs on a variety of computing platforms, and reported their own timings. For purposes of comparison, all speedups were calculated using eight processors for the parallel case.

Although the students did report development effort in hours, this data was sometimes spotty and inconsistent. For the sake of consistency with the procedure used for the NPB, effort was again measured in terms of SLOC, which were reported by an automated tool other than SLOCcount. The relative SLOC was calculated by dividing the SLOC for each student's parallel code by the SLOC for that same student's serial code.

The development time productivity was calculated by dividing the speedup by the relative SLOC for each student's code submission. The results of this analysis are presented in Section 3.3.

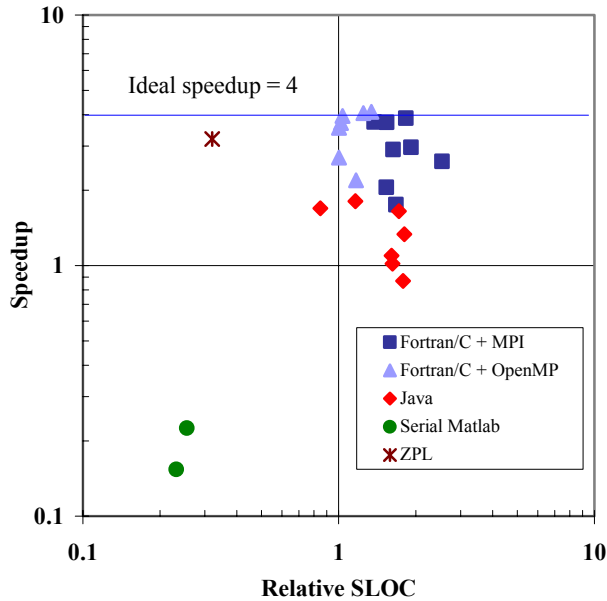


Figure 1. Speedup vs. Relative SLOC for the NPB

### 3. RESULTS

#### 3.1. NAS PARALLEL BENCHMARKS

Figure 1 presents a log-log plot of speedup vs. relative SLOC for the NPB. Each data point corresponds to one of the eight benchmarks included in the NPB suite, and the results are grouped by language (performance data was not available for some of the implementations). The speedup and relative SLOC for each benchmark implementation are calculated with respect to a reference serial code implemented in Fortran or C.

Each parallel code was run using four processors, setting an upper bound for speedup as indicated on the graph. For this study, no attempt was made to optimize or configure these benchmarks for the computing platform used. The goal of this study was not to judge suitability of one language over another for a given benchmark, but to observe general trends for a given language.

For example, the OpenMP implementations tend to yield parallel speedup comparable to MPI, while requiring less relative SLOC (Figure 1). This is reflected in the higher development time productivity values for OpenMP (Figure 2). As a general rule, we expect to see traditional parallel languages and libraries such as MPI and OpenMP fall in the upper-right quadrant of the graph. This reinforces our intuition that parallel performance is achieved at the cost of additional effort (over serial implementation).

The lone ZPL implementation falls in the upper-left quadrant of the graph, having a relatively high speedup and low SLOC count, as compared to the serial Fortran implementation. Although a single data point does not constitute a trend, this result was included as an example of an implementation that falls in this region of the graph. Accordingly, this ZPL implementation has a high development time productivity value (see Figure 2).

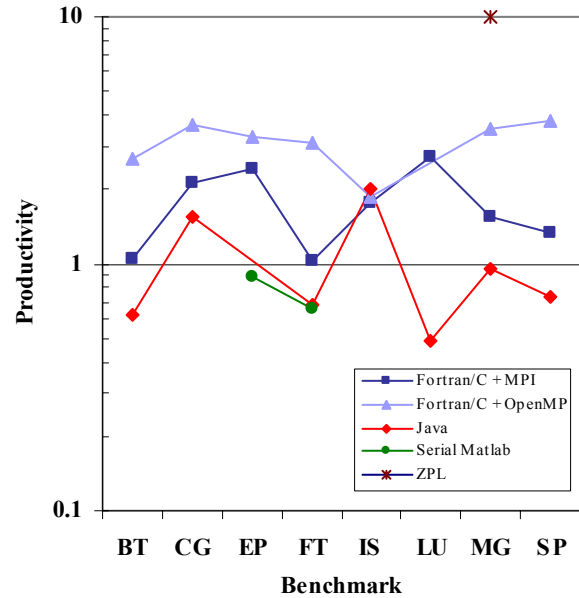


Figure 2. Development Time Productivity for the NPB

The Matlab results provide an example of an implementation that falls in the lower-left quadrant of the graph, meaning that its runtime is slower than serial Fortran, but it also requires fewer SLOC than Fortran (Figure 1). In fact, because of its low SLOC relative to serial Fortran, the serial Matlab manages to have a development time productivity value comparable to parallel Java (Figure 2).

A few of the Java implementations are in or near the lower-right quadrant of the graph. This indicates that, although additional lines of code were added to create the parallel implementation, little if any parallel speedup was realized. There may be any number of reasons why these implementations did not fare well – it should be noted that the Java implementations were generated via a semi-automated translation from the serial Fortran. In any case, those implementations that are in or near the lower-right quadrant will have development time productivity values at or below the baseline established by the serial implementation.

#### 3.2. HPC CHALLENGE

Figure 3 presents the results for the HPC Challenge benchmarks. The speedup and relative SLOC for each implementation were calculated with respect to a serial C/Fortran implementation. The parallel codes were all run using 64 dual-processor nodes, for a total of 128 CPUs. The implementations used for the Random Access benchmark (designated as RA in Figure 3) require a great deal of inter-processor communication, and so actually run slower as more processors are involved in a network cluster.

With the exception of Random Access, the MPI implementations all fall into the upper-right quadrant of the graph, indicating that they deliver some level of parallel speedup, while requiring more SLOC than the serial code. As expected the serial Matlab implementations do not deliver any speedup, but do all require fewer SLOC than the serial code. The pMatlab implementations

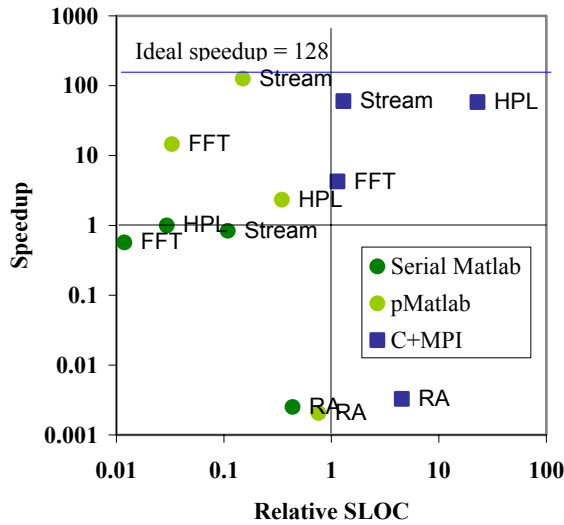


Figure 3. Speedup vs. Relative SLOC, HPC Challenge

(except Random Access) fall into the upper-left quadrant of the graph, delivering parallel speedup while requiring fewer SLOC.

The combination of parallel speedup and reduced SLOC means that the pMatlab implementations have higher development time productivity values (Figure 4). On average the serial Matlab implementations come in second, due to their low SLOC.

The MPI implementations, while delivering better speedup, have much higher SLOC, leading to lower development time productivity values.

### 3.3. CLASSROOM ASSIGNMENTS

Figure 5 presents speedup vs. relative SLOC results for a series of classroom assignments. The speedup and relative SLOC were collected for each student, and the median values for each assignment are plotted on the graph. As indicated on the graph, the ideal speedup in this case is eight, based on the use of eight processors for the classroom assignments.

Some of the assignments had median speedup values outside the range of 0.1 – 10. It is assumed that such outlier data is erroneous, and not representative of actual achieved performance with correct implementations. For the sake of clarity and comparison with the NPB results, the axes ranges are limited to 0.1 – 10, excluding some data points. Error bars indicate one standard deviation from the median value.

The MPI data points for the most part fall in the upper-right quadrant of the graph, resulting in development time productivity values at or above one (Figure 6). There was one MPI assignment in which most of the students were not able to achieve speedup, and this resulted in a median development time productivity less than one.

The OpenMP data points indicate a higher achieved speedup compared to MPI, while also requiring fewer lines of code

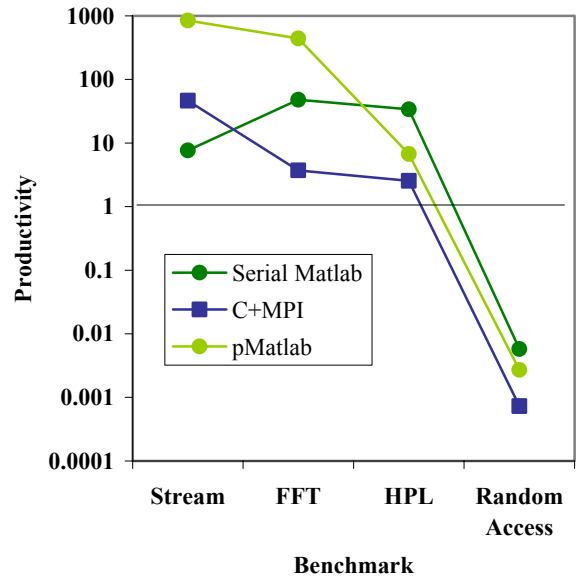


Figure 4. Development Time Productivity, HPC Challenge

(Figure 5). This leads to higher development time productivity values for OpenMP (Figure 6). Actually, the OpenMP assignment with a median SLOC less than the serial SLOC resulted in a development time productivity value greater than 10 (not shown).

The lone StarP data point has a median speedup value below one (slower than serial C), while requiring fewer lines of code than MPI (Figure 5). Due to the low speedup value, the StarP assignment has a development time productivity value less than one (Figure 6).

### 4. CONCLUSIONS

We have introduced a common metric for measuring development time productivity of HPC software development. The development time productivity formula has been applied to data from benchmark codes and classroom experiments, with consistent results.

In general the data supports the theory that MPI implementations yield good speedup but have a higher relative SLOC than other implementations. OpenMP generally provides speedup comparable to MPI, but requires fewer SLOC. This leads to higher development time productivity values. There are questions of scalability with regard to OpenMP that are not addressed by this study.

The pMatlab implementations of HPC Challenge provide an example of a language that can yield good speedup for some problems, while requiring fewer relative SLOC, again leading to higher values of the development time productivity metric.

In addition to discovering general trends for a given language, in practice this technique could be used to evaluate the productivity of programming models provided with two or more HPC systems, as part of the decision process associated with procurement. Consideration of the development time productivity metric, along

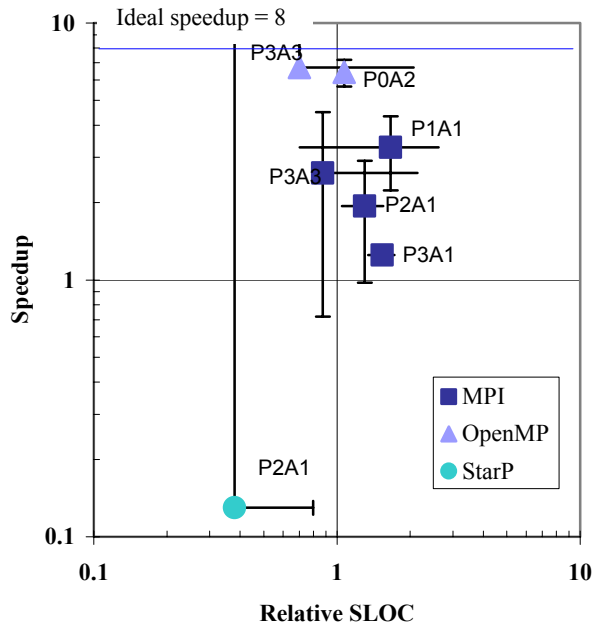


Figure 5. Speedup vs. Relative SLOC, Classroom assignments

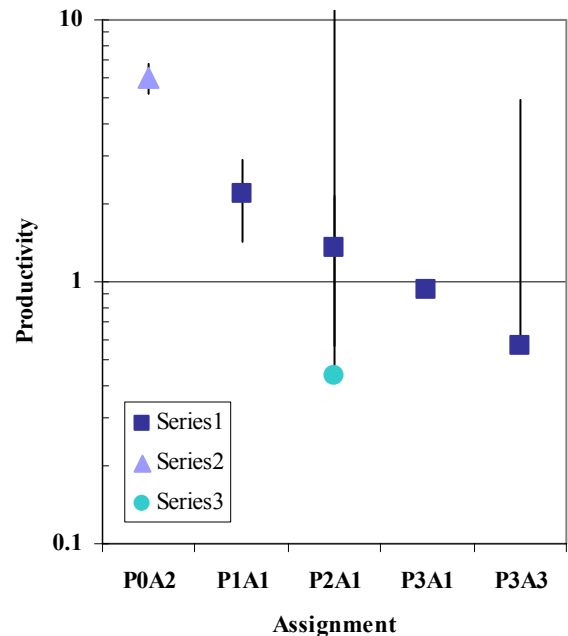


Figure 6. Development Productivity, Classroom assignments

with hardware performance and other factors, would give a more complete picture of overall system productivity.

Follow-on studies will examine more benchmark codes and language implementations. In the HPCS program there is an ongoing effort to collect a wide variety of HPC benchmarks implemented in as many languages as possible. Having this range of data will enable us to make more thorough comparisons between languages.

Further classroom experiments are planned, and as more data is collected it will be analyzed in the same manner, to see if other trends emerge. In addition to SLOC, effort data will be collected both automatically and by student reporting. Having two effort data sources will allow us to judge the accuracy of student reporting, as well as to fine-tune the automated collection process. This data will also enable us to further explore the relationship between effort and SLOC.

## 5. ACKNOWLEDGMENTS

We wish to thank all of the professors whose students participated in this study, including Jeff Hollingsworth, Alan Sussman, and Uzi Vishkin of the University of Maryland, Alan Edelman of MIT, John Gilbert of UCSB, Mary Hall of USC, and Allan Snaveley of UCSD.

## 6. REFERENCES

- [1] High Productivity Computer Systems <http://www.HighProductivity.org>
- [2] Kepner, J. "HPC Productivity Model Synthesis." *IJHPCA Special Issue on HPC Productivity*, Vol. 18, No. 4, SAGE 2004
- [3] Humphrey, W. S. *A Discipline for Software Engineering*. Addison-Wesley, USA, 1995
- [4] NAS Parallel Benchmarks. <http://www.nas.nasa.gov/Software/NPB/>
- [5] ZPL. <http://www.cs.washington.edu/research/zpl/home/>
- [6] Wheeler, D. SLOCcount. <http://www.dwheeler.com/sloccount/>
- [7] HPC Challenge. <http://icl.cs.utk.edu/hpcc/>
- [8] Haney, R. et. al. "pMatlab Takes the HPC Challenge." Poster presented at High Performance Embedded Computing (HPEC) workshop, Lexington, MA. 28-30 Sept. 2004
- [9] Choy, R. and Edelman, A. *MATLAB\*P 2.0: A unified parallel MATLAB*. MIT DSpace, Computer Science collection, Jan. 2003. <http://hdl.handle.net/1721.1/3687>