Towards a Timed Markov Process Model of Software Development

Burton Smith and David Mizell
Cray Inc.
Seattle, WA
burton@cray.com
dmizell@cray.com

John Gilbert and Viral Shah
UC Santa Barbara
Santa Babara, CA
gilbert@cs.ucsb.edu
shah@cs.ucsb.edu

ABSTRACT
The concept of a timed Markov model is introduced and proposed as a means of analyzing software development workflows. A simple model of a single researcher developing a new code for a high performance computer system is proposed, and data from a classroom experiment in programming a high performance system is fitted to the model.

Categories and Subject Descriptors
D.2.8 [**Metrics**]: Process metrics

General Terms
Management, performance, experimentation.

Keywords
High performance computing, productivity, software development, Markov model, signal flow graphs

1. INTRODUCTION
As difficult as it is to predict the performance of a high performance computing (HPC) system on a given program, it is even more difficult to predict a programmer's productivity in developing, debugging and performance tuning an HPC code. There is great variability in the types of algorithms and applications designed for HPC execution, and much greater variability in the programmer's experience and perhaps innate ability to design, program and tune parallel software.

This work is aimed at finding a way to apply more than just intuition to the question of where there is leverage in HPC productivity. It attempts to apply a quantitative model to the HPC software development and refinement process.

In this paper, we introduce the *timed Markov process* modeling approach and apply it to a simple example, of a researcher developing a new algorithm for solving a computationally-intensive problem on an HPC system.

2. TIMED MARKOV PROCESSES
Software development processes can be viewed as iterative and probabilistic. They are iterative in the sense that a programmer often operates within a loop in the software development process: edit – compile – launch test run, for example. It is probabilistic in that the times in each of the steps of the process can vary, and the time at which the programmer can move on to the next phase is unpredictable. A timed Markov process can model both aspects. We define a timed Markov process to be a Markov process that has dwell times, which can be random variables, associated with the states of the process and probabilities associated with the transitions to next states.

First Page Copyright Notice

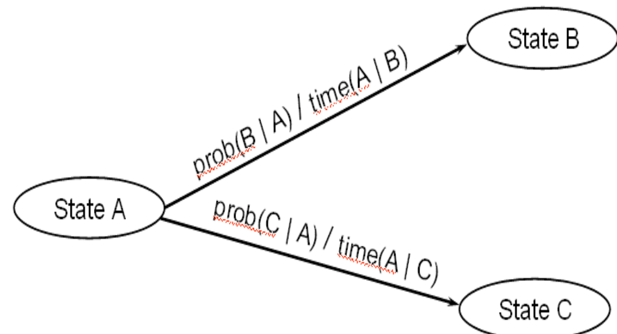

Figure 1. timed Markov process

In Figure 1, prob(B|A) is the probability of transitioning to state B given that the process is now in state A. time(A|B) is the dwell time spent in state A given that the next state transitioned to is B. prob(C|A), which would equal 1 – prob(B|A) in this example, is the probability of transitioning to state C given that the process is now in state A.

3. SIMPLE HPC SOFTWARE DEVELOPMENT EXAMPLE
The appeal of the timed Markov process (TMP) approach to quantitatively analyzing a software development workflow is its ability to estimate *time to solution*, the essential component of HPC productivity. The technique we use to estimate completion time of a TMP is based on the close resemblance they have to signal flow graphs [1]. Signal flow graphs have a set of reduction rules by which one can directly obtain the Fourier transform of the transfer function of the entire graph. These same reduction rules can be applied to a TMP, resulting in the transform of the expected completion time. Then the inverse Fourier transform gives the expected time to completion for the TMP. Eppinger and his colleagues have used similar techniques to model other iterative design processes [2]. In [3], Johnson and Brockman use Pareto techniques to analyze a similar graph of a software development process.

Our example is of a timed Markov process being used to represent the workflow of a single researcher developing a new algorithmic approach to solving some computationally-intensive problem, and implementing that algorithm on a high-performance system. The TMP graph for this workflow is shown in Figure 2. It represents our assumption that the researcher would begin by formulating a new algorithm for solving the problem of interest, and then write a program implementing the algorithm. Following that, the researcher would enter a correctness-debugging loop, around which the process would cycle until the program is deemed to be usefully free of programming errors. Next would be a performance-tuning loop, which would cycle until the program had been tuned enough that it gets adequate performance for large problems to be run on the HPC system.

In this example, for simplicity, we assume that the dwell time in each state is a constant, but that is not necessary. They could easily be made random variables, if one can reasonably assume that they are mutually independent. We also assume that there is no overlap between human activity and machine activity in the process, which is not true in practice but probably also not very significant. Finally, we ignore certain low-probability transitions, such as the researcher's getting exasperated at lack of progress in the performance-tuning cycle and decides to go back to "Square One" and reformulate the problem.

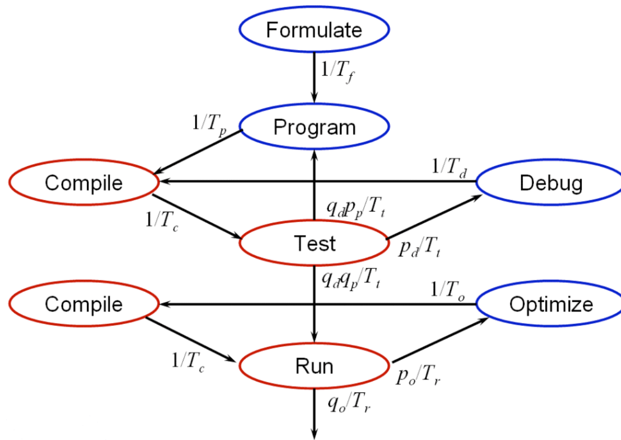

Figure 2. Researcher workflow model

In the researcher workflow model shown in Figure 2,

- T_f represents the time taken to formulate the new algorithmic approach.
- T_p is the time necessary to implement the new algorithm in a program.
- T_c is the compile time.
- T_t is the time necessary to run a test case during the debugging phase.
- T_d is the time the programmer takes to diagnose and correct the bug.
- T_r is the execution time for the performance tuning runs. This is the most obvious candidate of a constant that should be replaced by a random variable.
- T_O is the time the programmer takes to identify the performance bottleneck and program an intended improvement.
- p_p is the probability that debugging reveals a necessity to redesign the program.
- p_d is the probability that more debugging is necessary.
- p_O is the probability that more performance optimization is necessary.
- q_p, q_d, and q_O are $1 - p_p$, $1 - p_d$, and $1 - p_O$, respectively.

4. FITTING THE MODEL WITH EXPERIMENTAL DATA

The second author taught an Applied Parallel Computing class at UCSB in Spring 2004. This class participated in a productivity study, by completing a background questionnaire, a post-assignment questionnaire, and logging their activities in various ways during program development. In order to minimize intrusiveness, we also automatically collected several items of information relative to their progress in developing a parallel sorting code for the school's 64-node cluster. A snapshot of each student's code was collected every time he/she compiled. Because the students were using standard templates for input and output and standard (random) input data sets, any of these versions could be tested for correctness or performance. Thus, for each compile done by each student we had the following raw data:

- Date/time stamp
- Outcome: either "compile error", "runtime error", "wrong result", or "correct result"
- Running time, if "correct result"
- Number of lines of code

These were matched against the Figure 2 model using the following heuristics:

- Always begin in "program" (we couldn't measure "formulate").
- Compile errors stay in the same compile state.
- Run errors or wrong results stay in the compile1/test/debug loop.
- If a significant number of lines of code were added, go back to "program".
- The first correct result transitions to the compile2/run/optimize loop.
- Incorrect results in the optimize loop made us realize that the model needed another arrow, from "run" back to "debug".
- Usually the workflow ends in the optimize loop, but occasionally ends in the debug loop, necessitating another new transition in the model, presumably because the student, after getting a correct result, gave up during an attempt to debug an optimization.
- Dwell times were estimated from combinations of the timestamps, the measured run time of correct runs, and an assumption that every incorrect run took the same

amount of time and every compile took the same amount of time.

Figure 3 shows a revised version of the single researcher workflow model, after the results from the UCSB data collection had been incorporated. The new transitions indicated by the student data are shown in red.

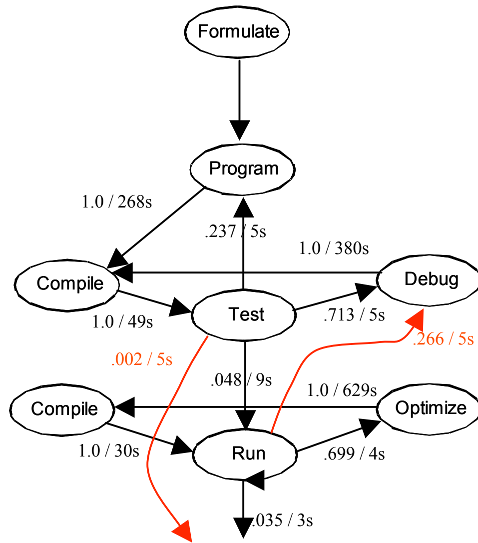

Figure 3. Revised workflow model

5. NEXT STEPS

The parallel computing class will be taught again at UCSB this spring. We are thinking about what data to collect, automatically whenever, possible, which would provide more accurate information we can use to try to fit the TMP.

Another interesting aspect of the TMP is that, once the expected time to solution is derived, sensitivity analyses can be performed on the time-to-solution equation, to estimate where the leverage is, when the goal is to improve HPC software development productivity. For Cray, that is the end goal of this analytical approach – to have a better idea of where to invest our resources to improve HPC software development productivity.

6. REFERENCES

[1] Howard, R., *Dynamic probabilistic systems*, John Wiley, New York, 1971.

[2] Eppinger, S., Nukala, M.and Whitney, D., Generalized models of design iteration using signal flow graphs, *Research in Engineering Design*, Vol. 9, No. 2, pp. 112-123, 1997.

[3] Johnson, E. and Brockman, J., Measurement and analysis of sequential design processes, *ACM Transactions on Design Automation of Electronic Systems*, Vol. 3(1), January 1998, pp. 1-20.