

# Modeling programmer workflows with Timed Markov Models

Andrew Funk\*      John R. Gilbert†      David Mizell‡      Viral Shah†

May 20, 2007

## Abstract

Software development is a complex process. Many factors influence programmer productivity – experience, choice of programming language, etc. – but comparisons of their effects are primarily anecdotal. We describe a quantitative method to capture programmer workflows using timed Markov models. We fit data collected from programmers in two separate classroom experiments to these timed Markov models, and compare the workflows of programmers using UPC and C/MPI.

## 1 Introduction

Higher level languages such MATLAB are often thought to be more productive to program in than Fortran or C. PGAS (partitioned global address space) languages are believed to be easier to use than message passing environments. There are several other widely held beliefs about programming and productivity: a well designed IDE might be more productive than command line tools, a debugger might be more productive than debugging by printing, interactive programming environments might allow for quicker development than compiled languages, and so on.

Such hypotheses are often anecdotal – it is hard to prove or disprove them. It should be possible to confirm or refute hypotheses about programmer productivity with a reasonable model of programming workflows coupled with experimental evidence. Quantitative analysis is desirable, but very hard to get.

We believe that our work can lead to an ability to choose a programming environment based on quantitative evaluations instead of anecdotal evidence. Several studies, for example [1, 4, 9], compare different software engineering processes and programming environments in a variety of ways, mostly qualitative.

Programmers go through an identifiable, repeated process when developing programs, which can be characterized by a *directed graph workflow*. TMMs (timed Markov models or timed Markov processes) are one way to describe such directed graphs in a quantifiable manner. We describe a simple TMM which captures the workflows of programmers working alone on a specific problem. We then describe an experimental setup in which we instrument the student homeworks in the

---

\*Email: [afunk@ll.mit.edu](mailto:afunk@ll.mit.edu). MIT Lincoln Labs.

†Email: [{gilbert,viral}@cs.ucsb.edu](mailto:{gilbert,viral}@cs.ucsb.edu). Department of Computer Science, UC Santa Barbara. This work was partially supported by the Air Force Research Laboratories under agreement number AFRL F30602-02-1-0181 and by the Department of Energy under contract number DE-FG02-04ER25632.

‡Email: [dmizell@cray.com](mailto:dmizell@cray.com). Cray Inc.

parallel computing class at UC Santa Barbara. We also describe the tools we developed for instrumentation, modeling, and simulating different what-if scenarios in the modelled data. Using our model and tools, we compare the workflows of graduate students programming the same assignment in C/MPI [3] and UPC [2] – something which is not possible without a quantitative model and measurement tools.

We describe Timed Markov Processes in Section 2. In Section 3, we describe our modeling of programmer productivity with Timed Markov Models. Section 4 contains a description of our data collection methodology. We compare UPC and C/MPI data in Section 5. In Section 6, we talk about the various tools we are designing to allow anyone else to perform similar analysis. We finish with concluding remarks in Section 7.

## 2 Timed Markov Processes

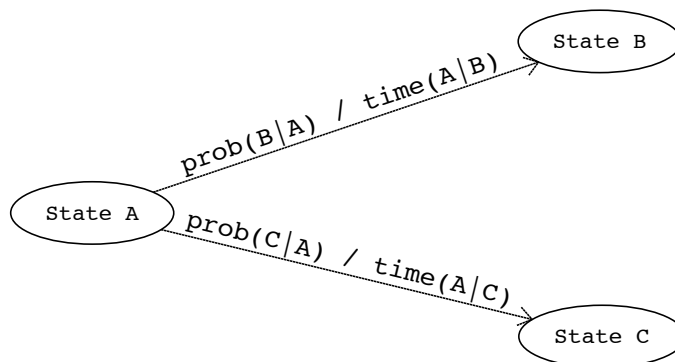


Figure 1: A timed Markov process

The process of software development is iterative and probabilistic. It is iterative in the sense that a programmer often repeats a sequence of steps 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 number of times a cycle will be repeated before the programmer can move on to the next phase is unpredictable. A timed Markov process can model both aspects.

A timed Markov process is a Markov process, augmented with dwell times for state transitions. Each state transition has associated with it, a probability of transition, and a dwell time. Timed Markov processes closely resemble signal flow graphs [6], for which well known methods exist to estimate time to completion. Iterative design processes and software development processes have been studied using similar techniques [7, 13].

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 Timed Markov models of programmer workflows

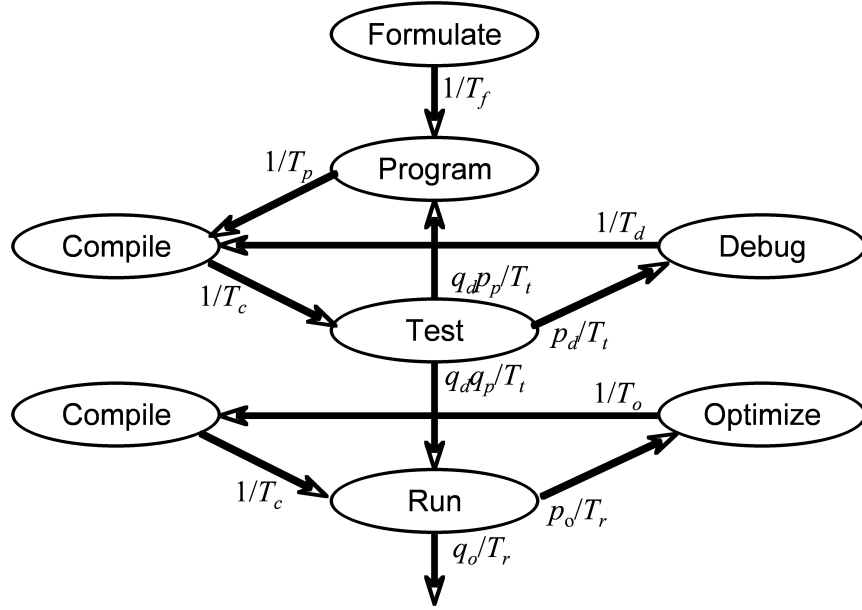


Figure 2: Lone programmer workflow

In our earlier work [12], we hypothesize a simple timed Markov model of a researcher developing a new application. It represents our assumption that the researcher begins by formulating a new algorithm for solving the problem of interest, and then writes a program implementing the algorithm. Following that, the researcher enters a correctness–debugging loop, around which the process cycles until the program is deemed to be free of programming errors. Next is a performance–tuning loop, which cycles until the program has been tuned enough that it gets adequate performance for large problems to be run on the HPC system. This is the workflow shown in Figure 2. In this workflow:

- $T_f$  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.

This model can also be used to describe the workflow of graduate students programming homeworks in a parallel computing class. We instrumented their programs, collected workflow data and fit it to such a timed Markov model in our 2004 classroom experiment [12]. While fitting the experimental data to the model, we discovered that in addition to the transitions described above, there were two more transitions. A programmer may introduce or discover a bug while attempting to optimize the program. As a result, there is a transition from the **Run** state to the **Debug** state. Sometimes, the last run may not be successful, perhaps because of a failed attempt to optimize. In such a case, an earlier correct run is treated as the final program – all the data we present is from programs that were eventually correct. Hence, there is another transition from **Test** to **Finish**. Figure 3 shows the result of our analysis of the 2004 classroom experiment.

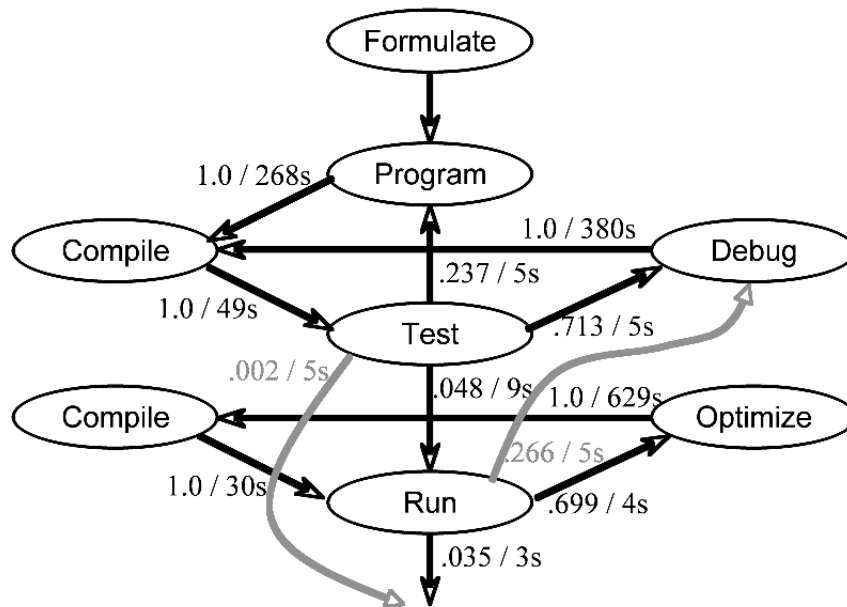


Figure 3: Fitting data from the 2004 classroom experiment to a timed Markov model.

## 4 Instrumentation and data collection

We used a comprehensive framework to collect data in our 2006 classroom experiment. We will describe our 2006 classroom experiment in full detail in a separate paper [11]. Briefly, the programmers programmed the *Game of Life* on a large grid, which would not fit in the memory of a typical desktop computer. The *Game of Life* is played on a two dimensional grid. Cells may be alive or dead – their state evolves according to set of simple rules based on the state of their neighbors. Half the programmers used C/MPI, while the other half used UPC.

Our data collection process gathers enough data at compile time and run time so that programmer experience can be accurately recreated offline. This allows us to replay the sequence of events

(every compile and run) and collect specific data that may be required by the modeling process but was not captured while the experiment was in progress. We used such replays very effectively for our 2004 classroom experiment [12]. Our replay capabilities were not perfect then, and we had to use reasonable proxies for run time parameters. We refined our techniques in light of the lessons learnt, and achieved perfect replay capabilities for the 2006 classroom experiment.

The programmers were provided with a basic harness which included build infrastructure, data generators, test cases and specific performance targets. We captured timestamps for every compile and run, stored every code snapshot at compile time, and recorded run time information such as number of processors, inputs, correctness and compute time. Due to a glitch in the run time data collection, some the run time information we needed for the modeling process was missing. However, we were able to gather the missing run time information by replaying every single compile and run for every programmer. The replays took 10 hours to compile, and 950 hours of processor time for all runs.

In our 2004 classroom experiment, the programmers wrote a parallel sorting code using C++/MPI. Since the model based on timed Markov processes was proposed after the 2004 classroom experiment, we used replays to gather the required data and fit it to a timed Markov model. Our experience with data gathering and modeling for the 2004 and 2006 classroom experiments has led us to believe that the *replay* is the most important facet of the data gathering process.

## 5 Comparing UPC and C/MPI workflows

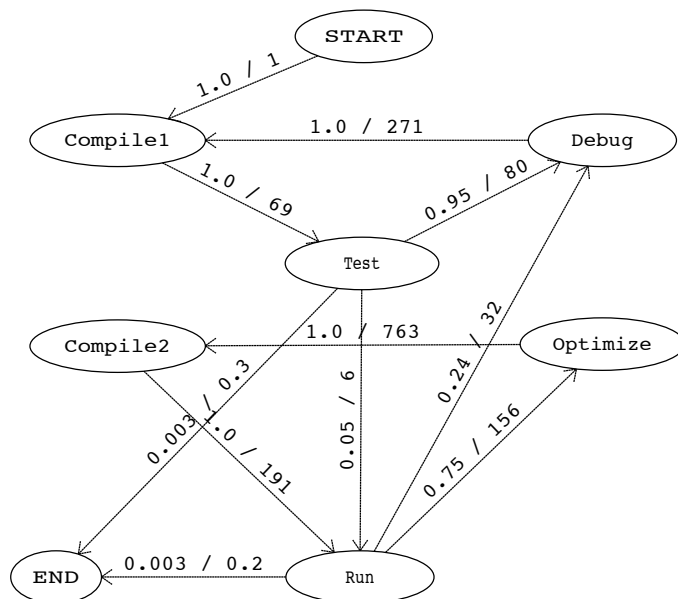


Figure 4: TMM fit to UPC workflow data. Edges representing state transitions are labelled as: probability of transition / dwell time in seconds.

Figure 4 shows the workflow of UPC programmers. Figure 5 shows that of C/MPI programmers. These diagrams of the TMMs were prepared using the TMMsim tool described in Section 6.2. This is a preliminary analysis with a small sample size (5 programmers using each language). Thus we do not attempt to draw final conclusions comparing the two languages. However, a number of aspects of these TMMs seem encouraging as regards the feasibility of this type of quantitative analysis.

First, the fitted transition times and probabilities from the 2006 classroom experiment are quite similar to those from the 2004 classroom experiment. Not surprisingly, most (92% to 95%) of “test” runs lead back into the debug cycle. We see that a “test” run is successful 8% of the time for C/MPI and 5% of the time for UPC; however, in the optimization cycle, 28% of C/MPI runs introduced new bugs compared to only 24% of UPC runs. It is not clear whether these differences are significant for this small sample size. A programmer spends much longer to attempt an optimization (763 seconds for UPC and 883 seconds for C/MPI) than to attempt to remove a bug (270–271 seconds). The time to optimize UPC (763 seconds) is smaller than that for MPI (883 seconds), suggesting perhaps that UPC optimization is carried out in a more small-granularity, rapid-feedback way.

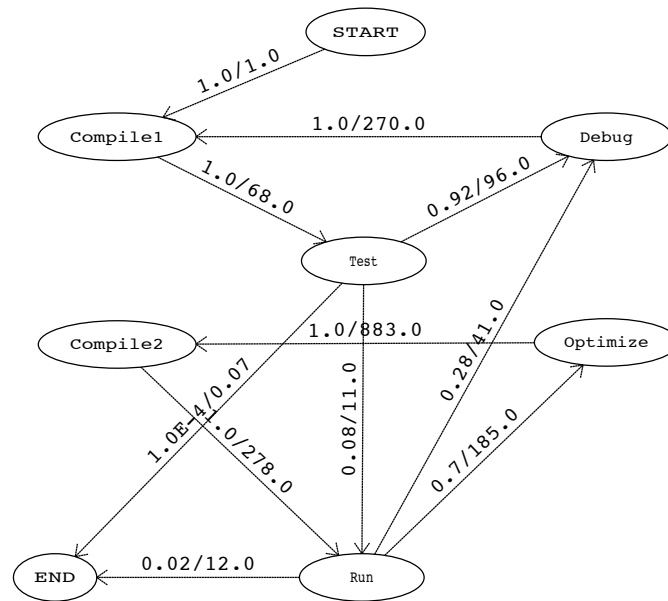


Figure 5: TMM fit to C/MPI workflow data. Edges representing state transitions are labelled as: probability of transition / dwell time in seconds.

## 6 Tools

The amount of data that needs to be analyzed to produce models of programmer workflows is quite large. We are developing automated tools for visualization, modeling, and simulation of TMMs to facilitate the kind of analysis described in earlier sections.

## 6.1 A tool for automatic TMM generation from collected data

There are two main types of data that are being collected in the experiments. Physical activities such as code edits, compiles, and executions are automatically captured by the instrumented development environment. During development, in some experiments, the students are also asked to record the time they spend performing logical activities such as thinking, serial coding, parallel coding, and testing. It is these logical activities that we use to create TMMs of the workflows. Alternatively, physical activities can be mapped to logical activities using a set of heuristics.

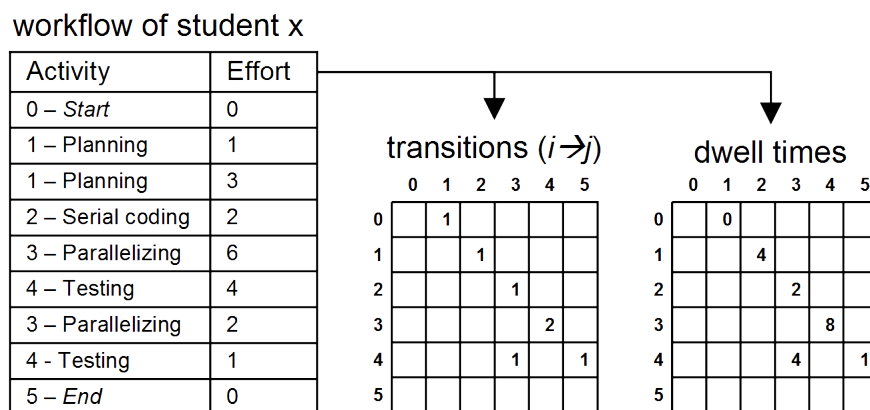


Figure 6: Mapping logical activities to a TMM

Whether the logical activities come from student logs [14] or heuristic mapping [12, 5], the end result is a list of activities and associated effort (measured in hours), as shown in Figure 6. We have created a Python program which parses this list of activities for each student, and counts the transitions and dwell times for each activity. In the example shown, the student starts in the planning stage and then transitions to serial coding. This is represented in the transition matrix as  $T_{12} = 1$ . Consecutive entries for the same activity are combined. Thus in the dwell time matrix, the amount of time spent in the planning state before transitioning to the serial coding state is represented as  $D_{12} = 1 + 3 = 4$ . These transitions and dwell times can be aggregated across students and similar assignments to create a larger sample for analysis.

We calculate the probability for each state transition from the transition matrix as:

$$prob(j|i) = \frac{T_{ij}}{\sum_j T_{ij}}$$

Similarly, the average dwell time for each transition is calculated as:

$$time(j|i) = D_{ij}/T_{ij}$$

Once the transition probabilities and dwell times have been computed, the next step is to generate a graph description that can be used to visualize the TMM. Our initial choice for visualization was the Graphviz tool, which uses the DOT language for graph description. Figure 7 shows the student workflow from Figure 6 visualized as a TMM using Graphviz. Using Graphviz we have created a graphical browser for rapid visualization of multiple data sets (see Figure 8).

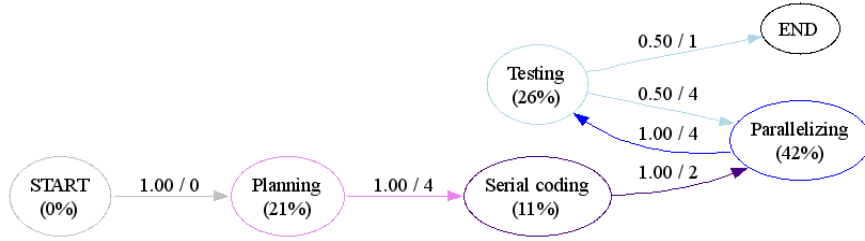


Figure 7: TMM visualization with Graphviz

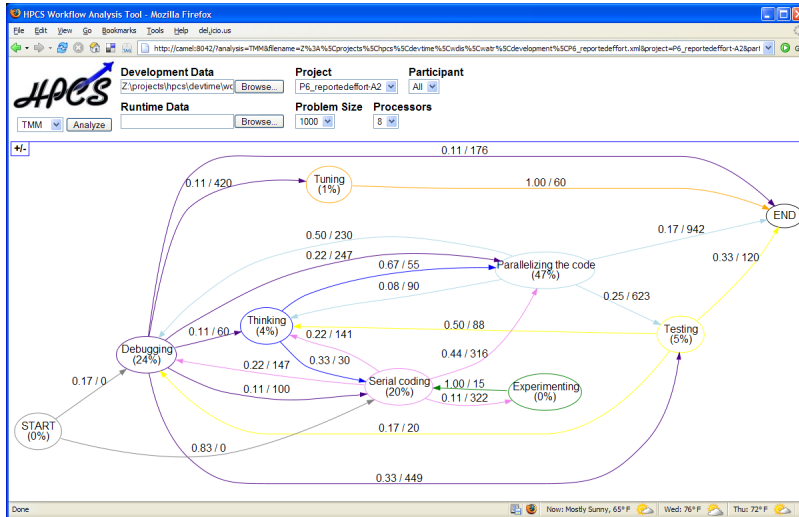


Figure 8: TMM visualization GUI

## 6.2 A tool for representing and simulating TMMs

Even with the help of automated tools, such as our prototype described in Section 6.1, the mapping from the activity data collected during a software productivity experiment to a workflow diagram and its corresponding Timed Markov Model will rarely be trivial. Programmers do not necessarily follow the exact same steps when developing or modifying code. The detailed actions programmers take can be abstracted to higher-level workflow steps in a variety of ways, some of which may turn out to be much more faithful models than others. Thus we expect that it may take several tries to obtain an acceptably accurate mapping from experimental data to a workflow.

We therefore wanted a tool with which we could:

- Draw and annotate new TMMs
- Import, view and modify previously defined TMMs such as those generated by the Python-based tool described in Section 6.1.
- Run discrete-event simulations of a TMM, in order to estimate the expected time through the workflow that it represents.

We based “TMMsim”, our TMM drawing and simulation software on VGJ [8], a Java-based tool (“Visualizing Graphs in Java”) for drawing directed and undirected graphs. The key feature

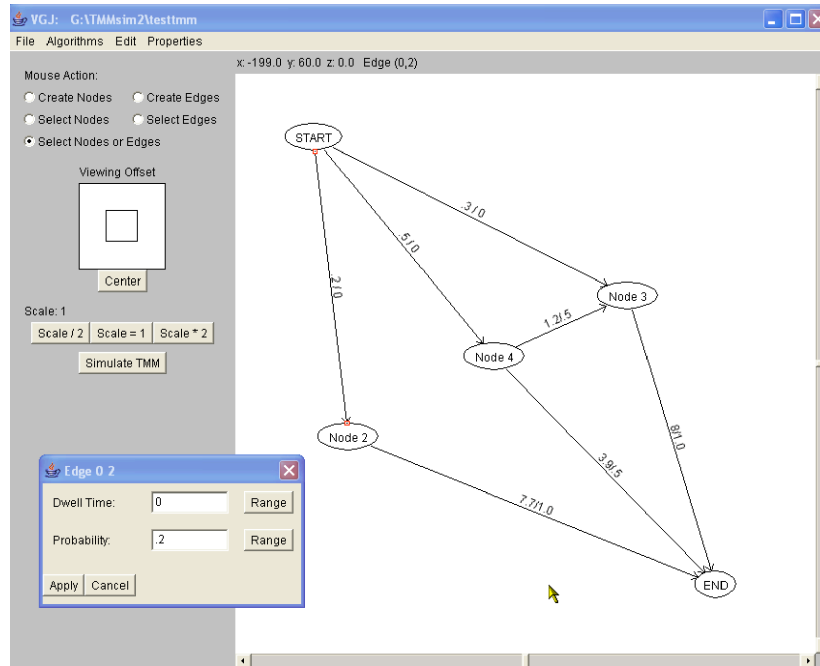


Figure 9: TMMsim, prototype tool for drawing, displaying, and simulating TMMs

we added was the ability to annotate each edge of the graph with a dwell time and a probability of departing the previous state via this edge, as required by a timed Markov model. The tool uses a slightly extended version of the Graph Modeling Language (GML) [10] to input previously-defined TMMs, including those generated by our TMM generation tool, (Section 6.1). Simple simulation code calculates the average time through the TMM workflow of 100,000 instances of its use. Figure 9 shows the TMM viewer/simulators user interface, along with a dialog box that has been opened in order to annotate the edge of an example TMM with a dwell time and a transition probability. The “Range” buttons in the edge annotation dialog allow a sequence of values to be used instead of a single number for the dwell time or the probability, causing a simulation average to be generated for each of the values, for the sake of sensitivity analysis.

## 7 Conclusion

We believe that programmers go through an identifiable, repeated process when developing programs, which can be characterized by a directed graph model such as timed Markov models.

We successfully gathered data and fit it to timed Markov models twice, in our 2004 and 2006 classroom experiments. The replay of programmer experience offline was one of the most important aspects of the data gathering process. The timed Markov models clearly indicate the most time intensive parts of the development cycle, quantitatively confirming our intuition – programmers spend most of their time debugging, and once they get a correct program, tuning it for performance is even more difficult. Our data also suggests, in this context, that programmers may introduce slightly fewer bugs in UPC programs, and find it easier to optimize them, as compared to C/MPI programs.

Clearly, this is only the beginning. A lot more data needs to be collected before languages can be compared in a meaningful way. Towards this end, we are building various tools for the community at large. These tools will provide a general framework for data collection and model construction to study programmer productivity in a variety of ways.

## References

- [1] Bradford L. Chamberlain, Steven J. Deitz, and Lawrence Snyder. A comparative study of the NAS MG benchmark across parallel languages and architectures. *Supercomputing '00: Proceedings of the 2000 ACM/IEEE conference on Supercomputing (CDROM)*, page 46, 2000.
- [2] The UPC Consortium. *UPC Language Specifications V1.2*, May 2005.
- [3] Jack J. Dongarra, Steve W. Otto, Marc Snir, and David Walker. A message passing standard for MPP and workstations. *Communications of the ACM*, 39(7):84–90, 1996.
- [4] Lorin Hochstein and Victor R. Basili. An empirical study to compare two parallel programming models. In *SPAA '06: Proceedings of the eighteenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 114–114, New York, NY, USA, 2006. ACM Press.
- [5] Lorin Hochstein, Victor R. Basili, Marvin V. Zelkowitz, Jeffrey K. Hollingsworth, and Jeff Carver. Combining self-reported and automatic data to improve programming effort measurement. In *ESEC/FSE-13: Proceedings of the 10th European software engineering conference held jointly with 13th ACM SIGSOFT international symposium on Foundations of software engineering*, pages 356–365, New York, NY, USA, 2005. ACM Press.
- [6] R. Howard. *Dynamic probabilistic systems*. John Wiley, New York, 1971.
- [7] E. W. Johnson and J. B. Brockman. Measurement and analysis of sequential design processes. *ACM Transactions on Design Automation of Electronic Systems*, 3(1):1–20, 1998.
- [8] Carolyn McCreary and Larry A. Barowski. VGJ: Visualizing graphs through Java. In *Graph Drawing*, pages 454–455, 1998.
- [9] Douglass Post and Ricard Kendall. Software project management and quality engineering, practices for complex, coupled multi-physics, massively parallel computational simulations: Lessons learned from ASCI. *Los Alamos National Laboratory*, 2003.
- [10] Marcus Raitner. The Graph Markup Language (GML) file format, 2001. <http://www.infosun.fmi.uni-passau.de/Graphlet/GML/index.html>.
- [11] Viral B. Shah, Vikram Aggarwal, and John R. Gilbert. Design of experiments in software engineering. Technical report, UC Santa Barbara, 2007.
- [12] Burton Smith, David Mizell, John Gilbert, and Viral Shah. Towards a timed Markov process model of software development. In *SE-HPCS '05: Proceedings of the second international workshop on Software engineering for high performance computing system applications*, pages 65–67, New York, NY, USA, 2005. ACM Press.
- [13] Smith, Robert P. and Eppinger, Steven D. Identifying controlling features of engineering design iteration. *Management Science*, 43(3):276–293, Mar 1997.
- [14] Marvin Zelkowitz, Victor Basili, Sima Asgari, Lorin Hochstein, Jeff Hollingsworth, and Taiga Nakamura. Measuring productivity on high performance computers. In *METRICS '05: Proceedings of the 11th IEEE International Software Metrics Symposium (METRICS'05)*, page 6, Washington, DC, USA, 2005. IEEE Computer Society.