lbsh: Pounding Science into the Command-Line

Eric Osterweil
UCLA
eoster@cs.ucla.edu

Lixia Zhang
UCLA
lixia@cs.ucla.edu

ABSTRACT

The definition of science is rooted in the notion that results must be reproducible. However, this must be more than just a presumption. Scientific disciplines must strive to ensure and facilitate that results actually *are* reproduced and build upon. In order to facilitate this in Computer Science, a detailed step-by-step record of how each result-set was generated is critical. This is called *Data Provenance* and when done well it enables researchers to understand the precise nature of each other's work. However, provenance is often only described at a high-level and this makes followon work, collaboration, and healthy scientific scrutiny more difficult. This is often because the complete documentation of all important processing steps is a very difficult task and is not always done. In this work we present a new open source utility called `lbsh` (pronounced "Pound-Shell") that provides a simple way for researchers to automatically log their work into virtual lab-books, automatically capture the provenance of their result-sets, query their lab-books for the provenance of individual result-sets, automatically re-run their prior work (with new or old inputs), and share this information with others via a very simple format. `lbsh` is publicly available, and we present 2 case-studies describing how it has already benefited actual Computer Science researchers.

1. INTRODUCTION

Scientific research requires that results be reproducible. Fields such as Biology and Physics not only mandate this, but their researchers routinely live by it as they reproduce findings from each others' published results. However, in Computer Science, the acceptance of results from measurement studies, simulations, and other general investigations is often predicated on some amount of faith. This is because the precise methods of derivation (or provenance) of data are generally not included with result-sets. One way to address this problem is to manually log all work done, but this has the immediate limitations that logging can impede exploration, and logs can contain both errors and irrelevant work. For example, there is a common mode of experimentation and exploration in which researchers evolve their approaches and process data at a very high rate. While in this mode, researchers may use combinations of an ever-growing set of different processing tools[1], and documenting each step can take as much effort as performing the actual experimentation. This proportion of extra overhead can be quite a burden on researchers. Furthermore, many experiments turn out to be irrelevant and the work for their result-sets should not be confused with work that has lead to meaningful results. Yet, knowing how a result-set was generated and how it can be reproduced is the hallmark of science.

In many natural sciences, provenance and reproducibility are facilitated by researchers' lab-books [10, 4]. Lab-books are very verbose logs of all steps taken during experimentation and are used to manually reconstruct experiments. While this approach is very useful it is prone to bloat and errors that arise from its highly manual nature. In a sense, this approach is too informal and loose. There are also a number of existing tools that attempt to address the problem of reproducibility by creating new environments and then mandating that users adjust their experimental practices accordingly. We view this general approach as being too restrictive. Provenance tools should not encumber researchers. Rather, researchers should have freedom in their methods of exploration and provenance tools should be able to seamlessly record their experimentation in enough detail to provide provenance for any specific results. As tools and experimental techniques evolve, a prescribed environment cannot be expected to play catchup at the expense of researchers who have sacrificed flexibility.

In this work we present a new tool called `lbsh` (pronounced "Pound-Shell")[2] that allows researchers to have the flexibility of using multiple independent tools and environments while fully and seamlessly documenting their experiments and meta-data. `lbsh` is so named because it creates an automated lab-book (`lb`) from within a user's shell (`sh`). It logs work without requiring users to change their behavior or their choice of environments and it implements facilities to automatically document and later query for the provenance of

[1]Data processing can be done in different shells (bash, tcsh), and with different utilities (`sed`, `awk`, `R`, `gnuplot`, etc.)

individual datasets.

lbsh capitalizes on 2 important insights: First, an abstract view of a repeatable experiment is that given a set of inputs and some processing, any experimenter should be able to get the same result-sets. Since, many Computer Science researchers do all of their processing and simulations in a command-line environment, automatically observing the inputs to experimentation is very tractable. By watching the command line, other meta-data (such as when commands are issued and file access patters), and a user's environment, an automated tool can provide data provenance and in many cases can automate the reproduction of results. Second, provenance tools must embrace users' heterogeneous environments and adjust to their experimental practices as a top priority. Mandating foreign environments or new languages for researchers is not a necessary precondition for scientific rigor.

lbsh has a simple design, is non-intrusive to users, and is very flexible. It abstracts the concepts of an *Experiment* and the provenance of result-sets (as *Provenance-Graphs*) and stores this information in a user's lbsh *Lab-Book*. Thus, with essentially no experimental overhead, a simple tarball containing result-sets, any custom scripts used, and a lbsh lab-book enables the community to inspect the detailed experimental provenance of a research team's work. This goal has already been achieved and we discuss this later in Section 5.1. lbsh is open source, is freely available via its website[2], and users are encouraged (though not required) to provide feedback via its online usage questionnaire, forums, and bug tracker.

The remainder of this paper is structured as follows: In Section 2 we describe existing work and approaches that address this type of problem. After surveying other approaches to this problem, we outline the design of lbsh in Section 3 followed in Section 4 by a description of its implementation. Next, in Section 5, we present some case studies of actual usage. Following this, in Section 6 we discuss our future work. Finally in Section 7, we conclude with a discussion of lbsh's benefits and some of its general limitations.

2. BACKGROUND

The definition of *data provenance* varies slightly throughout the literature. For the purpose of this work, we echo the definition used in [18]: "... we define data provenance as information that helps determine the derivation history of a data product, starting from its original sources." We define *result-sets* as just being dataset that represent the results of an experiment, and we use the term *meta-data* to very generally describe information that is used in the generation of result-sets.

The importance of reproducibility and scientific rigor has been very seriously addressed by most scientific disciplines through the use of lab-books [10, 4]. Generally, they are consulted when an experiment's results are fruitful, which can be quite some time after performing the actual work. Scientists often spend a great deal of effort manually documenting their experimentation so that results can be shared and reproduced. Understandably, being overly verbose is a far better choice than accidentally being too terse and losing track of key experimental details. However, keeping a manual lab-book is not necessarily a drop-in solution for Computer Scientists. For example, our experimentation is often very fast when compared to a biologist's. Logging steps for a biological experiment that may take months to run is a small overhead. By contrast, the overhead is much higher when logging the steps for an experiment takes the same amount of time as actually running it. Moreover, manually interpreting a lab-book can be quite laborious as one must sift through successful, unsuccessful, and irrelevant experiments to find the set of steps needed to reproduce any specific result-set.

2.1 Related Work

As natural sciences have begun to leverage computing resources for their research, more attention has been paid to documenting the experimental process of deriving datasets. In particular, much of this work has been done under the general heading of *eScience* or *Scientific Workflow*.

There exist several notable surveys of this type of work [18, 6, 20]. Several well-known systems in this field share a very general approach to running experiments while preserving meta-data and formally documenting an experimental process. Chimera[8], Kepler[5], myGrid[21], CMCS[11], and ESSW[9] implement workflow environments to allow scientists to specify their experiments. In these systems, experiments are defined by wrapping processing steps (and the data they use) in system-specific abstractions, manually mapping the dependencies between processing steps, and specifying meta-data.

The general approach of Scientific Workflow systems is to have scientists fully specify the steps involved in their experiments and to embed their processing in system-specific abstractions. Each of these systems require users to at least partially (and in some cases fully) input all meta-data for their experiments manually. After a lot of manual setup, then these approaches automate the actual running of experiments. These approaches assume that capturing the experiment's step-by-step procedure can only be done by tasking a researcher to specify it manually before the experiment is run, and their goal is to automate the actual running of experiments. This assumption motivates a tradeoff between automation and manual data input. As a result, the general drawback of using these systems is just that, users must abandon their normal processing environments and *use* these tools. While these systems have very powerful abstractions for formally modeling predefine experiments, they are not well suited for interactive exploration and documentation. This is because a researcher must spend a fair amount of effort specifying all the processing steps and meta-data for each experiment even those that take only moments to design and run and that turn out to be fruitless. Furthermore, these tools generally do not offer functionality to query for a dataset's provenance, or to aid in other post-processing of results. They are mainly focused on capturing the process of experimentation in order to facilitate re-running it.

`lbsh` adds a novel insight to the problem of recording data provenance. While Scientific Workflow systems strive to make work clearly documented and generally reproducible at the general expense of experimental freedom, `lbsh` recognizes that there are many times when this freedom is of paramount importance. Furthermore, even with this as the primary requirement, there is no reason that formal documentation cannot still be achieved. Rather than encumber researchers, we should have our computers do all of the work. Once `lbsh` has documented experiments, then it abstracts the dependencies and operations. In a sense, `lbsh` first does much of the diligence that these approaches pass onto their users, and then implements a similar dependency-based execution environment with the meta-data it captures automatically.

3. DESIGN

The design of `lbsh` is predicated on a few simple goals. Based on these goals, a streamlined set of simple abstractions are used to shape the design of `lbsh`.

To depict the usage of `lbsh` we consider a fictional user *Alice* who intends to do some investigation with some raw data she has obtained that is in a file called `raw.out`. *Alice* would like to keep track of her experiments using `lbsh` so that later she can disseminate her result-sets along with `raw.txt` and her `lbsh` lab-book as the provenance and vehicle for reproducibility of her results.

Alice's plan is to spend the next several weeks or months processing `raw.out` with her own scripts, commands such as R, gnuplot, etc. Her goal is to discover experimental evidence and produce some intriguing result-sets from her input data.

We will use this example to illustrate various abstractions and general points below.

3.1 Goals

The goals of `lbsh` are to provide a realistic tool that i) is immediately usable, ii) does not impinge on experimental practices, and iii) captures enough information and meta-data to make the derivation process of result-sets transparent, and reproducible.

In order to be immediately useful, `lbsh` must be able to perform its actions in whatever environment a user chooses to run. This includes cases where a user issues commands to their user shell [15, 12], any subshells such as R, Matlab, Perl, Python, etc. [14, 16, 7], or a combination of these environments. However, the choice of subprocess-tools is only one degree of freedom that users need.

There are many different ways in which users process their data. Freedom to choose processing techniques and environments must exist during all stages of experimentation. This is especially true during the early ad-hoc stages of a project in which users investigate multiple avenues, create new files, overwrite old files, and so on in rapid succession. It is, often, in these steps that discoveries are used to prompt follow-on experiments, and provenance tools must not hinder investigations.

However, while `lbsh` must be very low-overhead to use, it must also capture enough information and meta-data so that it is able to reconstruct provenance. This reconstruction should be easily manageable by a user and in an ideal case should automate the re-running of experiments. That is, users should be able to specify a set of results, see the experiments used to create them, and then have `lbsh` optionally re-run these experiments automatically from either the original data-sets (or with new data). In order to do this by monitoring the command line, `lbsh` must be able to recognize the difference between commands that are being executed, and cases where a user has entered a tool like a text editor. In these cases, `lbsh` should *not* record what a user has entered. These applications typically repaint the user's screen and we call this, "capturing the screen."

3.2 Abstractions

The abstractions in `lbsh`'s design are derived from logging users' command line behaviors and tracking their datasets (as they are created, used, and modified). Just as any other lab-book would contain steps taken in an experiment, `lbsh` contains the commands issued to the shell and any other command-line tools executed by the user. It abstracts groups of commands into *Experiments*, defines a user's set of experiments as a *Lab-Book*, and models dependencies between experiments (via their datasets) as *Provenance Graphs*.

Experiments: `lbsh` takes the perspective that users are the best judges of which sets of actions are atomic. The experiment abstraction is designed to let users specify a set of commands that logically work together in some way. Users specify an experiment by signaling `lbsh` to begin logging and optionally telling it the name of the new experiment. Then, it begins logging commands as they are entered and tracking a user's environment and data files for meta-data.

For example, suppose *Alice* starts her day of experimentation by logging into a server, spawning `lbsh`, and then thinking about what kinds of experiments she should run. Then, when she is ready to start exploring, she instructs `lbsh` to start logging a new experiment. When she has finished with the set of steps for this experiment she concludes it. Later, when she starts a new experiment, she instructs `lbsh` to start logging again. This has allowed her to log her work, and implicitly organize the steps into separate experiments.

`lbsh` does not have any need to understand the nature of an experiment, it is just a way for users to organize their work. This is actually a large benefit of `lbsh` because it makes its experimental framework very general and useful in almost arbitrary types of work, which we discuss in Section 5.2.

Tracking the environment helps inform the behavior of the commands, and watching file access and modification times helps understand the dependencies between experiments. There is no explicit reason that forbids users from starting work in one experiment and finishing it an another. The experimental abstraction is intended to model atomic experiments, but `lbsh` does not mandate this.

Lab-Book: Each experiment is logged separately into `lbsh`'s lab-book. The lab-book is a logical set of all experiments that a user has run. This abstraction is a simple way of helping users isolate work on different projects (i.e. creating a different lab-book for each project), and is also useful when defining provenance graphs.

To illustrate this, consider our user *Alice*; every time she runs a new experiment it is logged separately, but if she wants to manually (or automatically) look at her own work, it is all organized in a single place. Later, when she moves onto another project, she can save her current lab-book elsewhere and start a new one.

Provenance Graphs: This abstraction addresses the challenge of recalling the exclusive set of experiments needed to recreate specific result-sets and their dependencies. While an individual experiment may be traceable, knowing which prior experiments were used to generate the input files to a specific experiment may take some time to manually discover.

Suppose our user, *Alice*, wants a way to interrogate her `lbsh` lab-book about *which* experiments were used to generate some of her result-sets. Specifically, suppose that *Alice* has some statistical correlations and a set of graphs that she is happy with. She may then want to know how she arrived at them (their provenance). This could be after several days, weeks, or months of other work. At this point, she uses `lbsh` to create a *Provenance Graph* that shows all the relevant experiments and how they depend on each other. A visual representation of an instance of this is seen in Figure 1. Here *Alice* has 6 related experiments. Based on their names, we see that *Alice* ran an initial experiment and did some followup work that must have used some files produced by the "Initial experiment." Later, she ran another experiment and its name indicates that some new data was found and an experiment was run on it. This could mean that *Alice* now uses more than just `raw.txt` to get her results. In her next experiment, *Alice* does some processing that depends on data from 2 previously unrelated experiments and she called it "Merging Data." Following this, 2 separate experiments use data produced by this "Merging Data" experiment, but don't depend on each other at all.

Formally, we define a provenance graph as a Directed Acyclic Graph (DAG) $G = (V, E)$ that contains the entire set of experiments that are relevant to a result-set (and their dependencies). We define an experiment as a node v_i and a user's lab-book as a set of experiments V where $v_i \in V$. The file access and modification times are used to discern dependencies between experiments. If one experiment v_i creates or just modifies a file f, and a subsequent experiment v_j reads or uses f, then v_j depends on v_i. Formally we model this dependency as: $e_k = (v_i, v_j)$. In G, each dependency is an edge, $e_i \in E$ where $e_i = (v_j, v_k)$ and $j \neq k$. The causal ordering in G is determined by the start-date of each node, and this ensures that G is acyclic.

The utility of provenance graphs is to formally define which experiments depend on each other and what order they were run in. The intention is to allow a user to clearly see how datasets were derived, and potentially re-run them in order.

4. IMPLEMENTATION

`lbsh`'s implementation: i) provides a flexible and easy-to-use environment, ii) lets the user define the beginning and end of each atomic experiment, and iii) provides a suite of tools that can describe the provenance of result-sets and re-run experiments.

The implementation is split into a user-shell monitor (`lbsh`) that logs commands and meta-data into the automated lab-book and a set of Perl modules that facilitate the processing of the lab-book. `lbsh`'s behavior is specified by a configuration file (`.lbshrc`) in the user's home directory. It reads the user's "lab-book" directory from this configuration file and stores each experiment there. Each experiment is given an ID that corresponds to the UTC timestamp from the starting point. Though this is not globally unique across users, it is anticipated that a single user's lab-book will only have 1 experiment for any given second. The `lbsh` monitor is written in C++, it uses the Boost [17] regular expression library, and it has been tested and deployed on Linux, OS X, and FreeBSD. When `lbsh` is invoked, it immediately

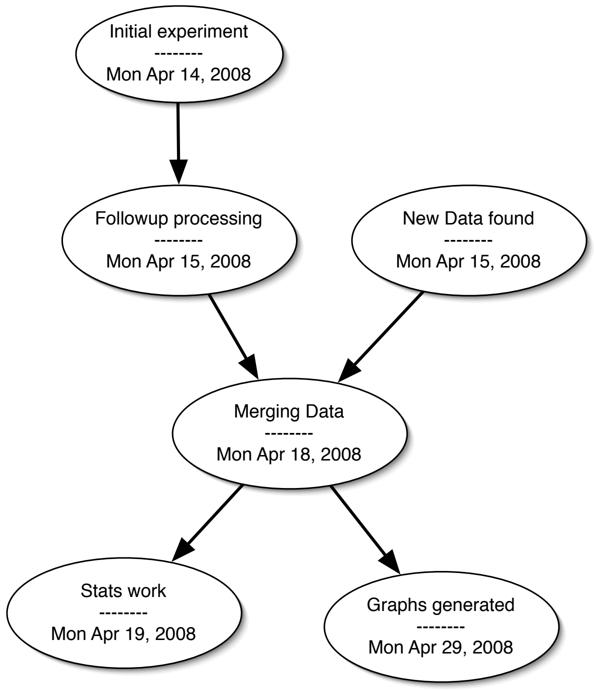


Figure 1: An example Provenance Graph showing several experiments described by their user-supplied descriptions and linked based on file usage and modification times.

spawns a new *worker* shell that is responsible for executing all user commands. The worker shell is the same type as the user's current shell.

Figure 3 depicts the general flow and how `lbsh` is able to receive commands from the user, pass them to the worker shell, and then record what the worker shell returns. Users specify the beginning of an experiment by issuing the meta-sequence `ctrl-b`. Information about every experiment is stored into 3 new files: a *script file* with user commands in it, a *meta file* with both general meta-data and per-command meta-data, and a *stat file* with a list of files accessed and modified during the experiment. The implementation has 4 main components: the *I/O State Engine*, the *Line Interpreter*, the *Experiment Logger*, and the *File Monitor*.

To assist researchers in processing their `lbsh` labbooks and doing general post processing (like re-running experiments), `lbsh` comes with a set of general Perl modules and 2 Perl scripts: *file-provenance.pl* and *exeggutor.pl*. These 2 scripts use the `.lbshrc` and a labbook to output a provenance graph and re-run experiments, respectively.

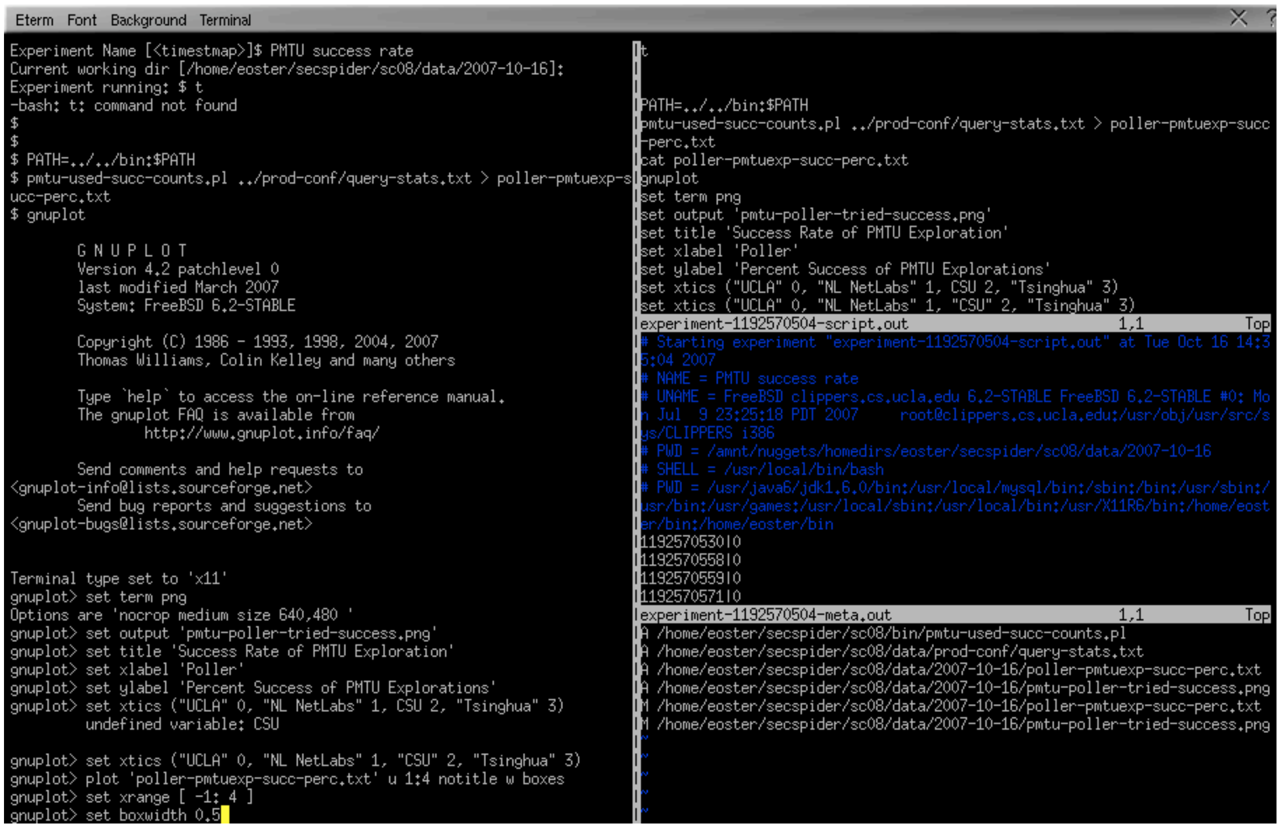
Files: An example of `lbsh`'s 3 types of files, and the corresponding session that created them can be seen in Figure 2. The script file contains the commands run by the worker shell. This file includes all commands including those that spawn other processes. When a process has "captured the screen" (like a text editor) this file does not capture any more output until the process has released the screen. We describe this concept in more detail below.

The meta file begins with a header of general meta-data from the user's environment. This includes, the start time of an experiment, the current working directory, the optional name of the experiment (provided by the user), the shell in use, the PATH environment variable, and all of the `uname` information about the machine (its operating system, version, etc.). After the header, the meta file has an entry for each command issued during an experiment. The information logged is a timestamp of when each command was issued, and a boolean indicating if the command subsequently captured the screen ('1') or not ('0'). This boolean is useful when re-running an experiment from a script file. If a command captured the screen, then an automated script should not automatically re-run that command. The final line of the meta file is the time at which the experiment was concluded.

The stat file contains a list of file names. Each file is on a separate line that begins with an 'A' if the file was accessed at any time during the experiment, or an 'M' if it was modified (or created) during the experiment. The files monitored are only those in the user's *data directory*, which is specified in the user's configuration file.

Components: `lbsh` uses the text output from its worker shell to record user commands that are executed. It uses the line echoed, and not the text typed, in order to properly capture commands that are issued by using a shell's history, tab-complete, or other mechanisms. This is because a user may not have actually *typed* the command that was echoed (and run) by the shell. This technique also allows `lbsh` to continue logging commands even when a user spawns a new subprocess-shell (such as R, Matlab, GnuPlot, Perl, Python, etc).

In addition to being able to differentiate between tools that have "captured the screen," and command-line directives `lbsh` must also understand the difference between a user command and any other output from a shell, such as the output that is sent after issuing a command. To do this differentiation `lbsh` implements a simple I/O state engine. This I/O state engine is completely passive and takes no action until a user starts an experiment. During experiments, the state engine reads input from the user's original shell and passes it to the newly spawned shell (the new shell performs all of the work). All output from the worker shell is collected first by the I/O state engine and then given to the line interpreter. If the user has pressed enter, then the line interpreter is interrogated by the I/O state logic to determine if the output should be logged or the screen



```
Experiment Name [<timestmap>]$ PMTU success rate
Current working dir [/home/eoster/secspider/sc08/data/2007-10-16]:
Experiment running: $ t
-bash: t: command not found
$
$
$ PATH=../../bin:$PATH
$ pmtu-used-succ-counts.pl ../prod-conf/query-stats.txt > poller-pmtuexp-s
ucc-perc.txt
$ gnuplot

        G N U P L O T
        Version 4.2 patchlevel 0
        last modified March 2007
        System: FreeBSD 6.2-STABLE

        Copyright (C) 1986 - 1993, 1998, 2004, 2007
        Thomas Williams, Colin Kelley and many others

        Type `help` to access the on-line reference manual.
        The gnuplot FAQ is available from
                http://www.gnuplot.info/faq/

        Send comments and help requests to
<gnuplot-info@lists.sourceforge.net>
        Send bug reports and suggestions to
<gnuplot-bugs@lists.sourceforge.net>


Terminal type set to 'x11'
gnuplot> set term png
Options are 'nocrop medium size 640,480 '
gnuplot> set output 'pmtu-poller-tried-success.png'
gnuplot> set title 'Success Rate of PMTU Exploration'
gnuplot> set xlabel 'Poller'
gnuplot> set ylabel 'Percent Success of PMTU Explorations'
gnuplot> set xtics ("UCLA" 0, "NL NetLabs" 1, CSU 2, "Tsinghua" 3)
         undefined variable: CSU

gnuplot> set xtics ("UCLA" 0, "NL NetLabs" 1, "CSU" 2, "Tsinghua" 3)
gnuplot> plot 'poller-pmtuexp-succ-perc.txt' u 1:4 notitle w boxes
gnuplot> set xrange [ -1: 4 ]
gnuplot> set boxwidth 0.5
```

```
t


PATH=../../bin:$PATH
pmtu-used-succ-counts.pl ../prod-conf/query-stats.txt > poller-pmtuexp-succ
-perc.txt
cat poller-pmtuexp-succ-perc.txt
gnuplot
set term png
set output 'pmtu-poller-tried-success.png'
set title 'Success Rate of PMTU Exploration'
set xlabel 'Poller'
set ylabel 'Percent Success of PMTU Explorations'
set xtics ("UCLA" 0, "NL NetLabs" 1, CSU 2, "Tsinghua" 3)
set xtics ("UCLA" 0, "NL NetLabs" 1, "CSU" 2, "Tsinghua" 3)
experiment-1192570504-script.out                            1,1           Top
# Starting experiment "experiment-1192570504-script.out" at Tue Oct 16 14:3
5:04 2007
# NAME = PMTU success rate
# UNAME = FreeBSD clippers.cs.ucla.edu 6.2-STABLE FreeBSD 6.2-STABLE #0: Mo
n Jul  9 23:25:18 PDT 2007     root@clippers.cs.ucla.edu:/usr/obj/usr/src/s
ys/CLIPPERS i386
# PWD = /amnt/nuggets/homedirs/eoster/secspider/sc08/data/2007-10-16
# SHELL = /usr/local/bin/bash
# PWD = /usr/java6/jdk1.6.0/bin:/usr/local/mysql/bin:/sbin:/bin:/usr/sbin:/
usr/bin:/usr/games:/usr/local/sbin:/usr/local/bin:/usr/X11R6/bin:/home/eost
er/bin:/home/eoster/bin
1192570530|0
1192570558|0
1192570559|0
1192570571|0
experiment-1192570504-meta.out                              1,1           Top
A /home/eoster/secspider/sc08/bin/pmtu-used-succ-counts.pl
A /home/eoster/secspider/sc08/data/prod-conf/query-stats.txt
A /home/eoster/secspider/sc08/data/2007-10-16/poller-pmtuexp-succ-perc.txt
A /home/eoster/secspider/sc08/data/2007-10-16/pmtu-poller-tried-success.png
M /home/eoster/secspider/sc08/data/2007-10-16/poller-pmtuexp-succ-perc.txt
M /home/eoster/secspider/sc08/data/2007-10-16/pmtu-poller-tried-success.png
~
~
~
~
~
```

Figure 2: This Figure shows a sample `lbsh` session (on the left). Here we can see that the user has typos, does some command-line work, and even spawns `gnuplot` for some interactive graphing. The resulting script file in the upper right corner has captured each command (including the typos) and has stripped off the prompts. Of particular note is that the commands are seamlessly recorded even as the user enters `gnuplot`. Below this is the meta file. In this file `lbsh` has recorded a great deal of meta-data including the starting working directory, the name of the experiment, the start time, etc. In addition, each command recorded in the script file has a corresponding entry that lists the time the command was issued and if it captured the screen. On the bottom right, the stat file lists the files that have been accessed and modified during this experiment.

was captured. If this was a command of interest, then the I/O state engine directs the experiment logger to record the current line, and prompts the file monitor to check the `atime`s and `mtime`s of files in the user's data directory.

The line interpreter is responsible for deciphering standard `termcap` [19] commands issued by the underlying shell, processing higher level semantics (such as identifying and stripping off prompts from the output), and determining if the output is from an interactive command-line process, or if the terminal was captured. The first task is a relatively straightforward operation. Users' environments map commands such as `ce` (or "clear everything") and "`\b \b`" (or delete) to special sequences. The line interpreter learns these mappings from the user's environment and then applies them to output from the worker shell. Next, the output will often (though not always) start with a prompt. When this is the case, this component strips prompts off so that the resulting lab-book entry only contains the user commands. This is done in large part for clarity, but also to aid the automated re-running of experiments. Finally, the line interpreter also inspects the output to determine if the screen has been captured. This determination is made by looking for termcap-style commands that reference absolute coordinates on the screen. If the output includes "random-access" coordinates, then the line interpreter assumes this is *not* a command-line application. However, this component does not take any actions based on its interpretation, it just maintains state.

After giving output to the line interpreter, the I/O state engine checks the line's state. If the line was a terminal command, then the state engine instructs the experiment logger to log the last line returned by the worker shell.

Post Processing: Based on the file access and modification times, `lbsh` can determine experimental provenance. For example, suppose *Alice* starts an experiment named "Day 1," and creates a set of files including one called `day1.out`. A few days later she begins a new experiment named "Day 4." In this new experiment she uses `day1.out` and then creates a new file: `day4.out`. Suppose a few weeks later she wants to know which experiments must be re-run to get `day4.out`. One thing that she must know is that the "Day 4" experiment depends on files from the "Day 1" experiment.

Alice can query for this by using `lbsh`'s Perl modules to parse and process the meta-data in her lab-book. She will issue the command:

```
file-provenance.pl -f day4.out
```

This will produce the ordered list of experiments needed to create this file. Then, for each experiment, she can issue the command:

```
exeggutor.pl <ID>
```

The result of these steps will be that `day4.out` will have been regenerated.

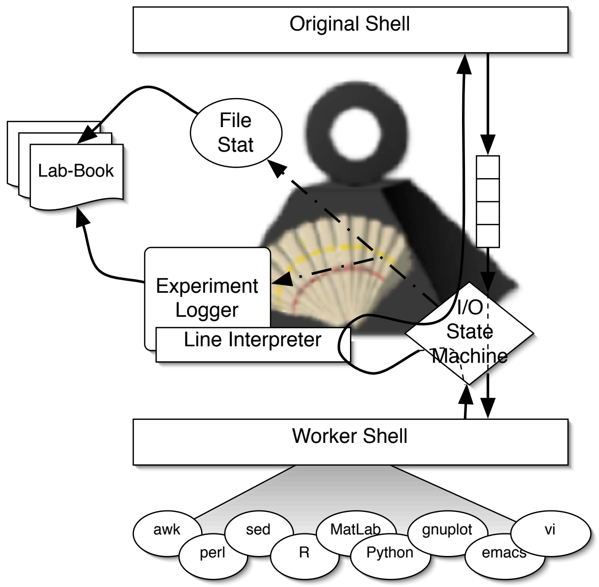


Figure 3: `lbsh` interposes between the original shell of a user and a newly spawned shell (of the same type). All key strokes are passed straight through until the user starts an experiment. Then, key strokes are passed through and resulting commands and meta-data are logged.

5. CASE STUDIES

`lbsh` has been in use among a set of researchers since October 2007. During this time, it has proven to be useful in a number of ways. In this Section, we detail 2 case studies that demonstrate ways in which it addresses general problems.

In the first case study, the user does not manually practice any significant diligence. The focus of this analysis will show the general benefit of `lbsh` during experimentation and through the process of an actual paper submission. On the opposite end of the spectrum, the second case study follows a user who manually maintains a complete lab-book of all processing steps and shows the problems that arise in this case and how `lbsh` was useful.

5.1 Case 1

In this case study, we follow a user who does not practice any manual recording of his experimentation. This user creates scripts, uses command-line tools such as `awk`, `sed`, `sort`, plots data, etc. Generally, after finishing with a round of experimentation, or after a submission, or even some time after that, the user has not given himself any rigorous way of recalling how result-sets and graphs were generated.

During October of 2007, this user began using `lbsh` to enhance his experimentation. He began with an initial data-set that was taken from a long-standing Internet monitoring project. On October 11^{th}, 2007 an initial set of data was taken and processing began. Over the course of weeks, result-sets were created, and a set of graphs were created and examined.

The findings were intriguing enough to prompt a paper submission, and experimentation evolved into more focused processing and a further refinement of some previous experiments. The results that showed the most merit became the foci of the data processing. However, being several months later, new data-sets were available and the comparison between results from October and later results prompted the need to re-run processing on newer data.

In the past, this user's lack of logging and general lack of diligence would have prompted a laborious process of trying to reconstruct the processing needed to reproduce result-sets, and then incorporate new data-sets. However, `lbsh` was able to not only list the experiments that were used to create the important result-sets, but it was also able to re-execute them automatically. By using `lbsh`'s lab-book and issuing the `file-provenance.pl` command, and then the `exeggutor.pl` command, the entire set of graphs and result-sets were regenerated. In order to be confident in the tool, this user actually used an automated script that created MD5 checksums on each old and new file to verify that the regenerated data was identical to the first time they were generated.

Next, the user wanted to re-run the processing but on new data. Based solely on the user's lab-book, `file-provenance.pl` was able to generate the entire set of relevant experiments and their dependency relationships. A depiction of this graph (automatically generated by `file-provenance.pl`) is seen in Figure 4. This figure shows that 26 experiments contributed to the results in this user's paper. By contrast, the user ran 90 total experiments from which `lbsh` only reran those needed. Re-running was very easy because the user just renamed the old processing directory from `/home/<user>/work` to `/home/<user>/work-bak`, created a new `/home/<user>/work` directory and seeded it with new data files that had the same names as the originals. Then, `exeggutor.pl` was invoked with the provenance graph as input and the new result-sets were automatically generated. The results of using new data prompted further experimentation which produced further results. By enabling the automation of data processing, `lbsh` was able to facilitate experimentation that may have otherwise been hampered by a lack of diligence, lack of time, or even manual errors involved in repeating work.

By re-running the same processing used on his initial datasets again on new datasets, the paper in question gained an entire new aspect. the 26 experiments were re-run to enhance the initial measurement study into a longitudinal measurement study. Though this would have been possible with a fair amount of manual effort, `lbsh` made it seamless.

Beyond the benefit of easy reconstruction of results, `lbsh` is also useful in that the entire provenance of result-sets is cleanly represented by a tarball that contains the i) lab-book directory, ii) the `.lbshrc` file, and iii) the input data sets and any user scripts needed. From these, the results of this paper are entirely transparent and reproducible. Furthermore, the results can be made public and subject to scientific scrutiny, and perhaps just as important is the fact that the user's experimental behavior did not need to be changed to support this.

5.2 Case 2

Our second case study follows a user who logically occupies the opposite part of the diligence-spectrum from our first user. This user maintains a detailed manual lab-book (a text file) of all commands needed to re-run the experiments conducted during a project. To maintain this, the user copy-and-pastes from his command-line to this text file. This user primarily uses the statistical analysis tool `R` [14], and as a result his commands are executed through a subprocess' shell (rather than [15]).

Before `lbsh`, this user would start from a raw data-set and perform operations that create R tables. These tables would then be combined into large hashes and are subjected to several rounds of detailed processing and analysis. The results would include data files and graphs.

The diligence of this user is admirable because as experiments are run, he constantly updates his manual lab-book and periodically verifies its accuracy by copy-and-pasting large sections from the lab-book into the shell to be sure that it executes properly. However, while his commitment is impressive, the approach demonstrates its general flaws.

In the past when this user wanted to know how certain files were generated he would perform textual searches through his lab-book. Often times, the same file name would appear multiple times but sometimes these entries actually corresponded to different files whose names collide, but whose directories were different. As a result, this user would have to manually inspect the commands leading up to the file names to determine which are the appropriate commands to run. In this case, the user did not actually want to re-run *all* the commands for the whole project just for any single file.

Due to the fact that the manual lab-book has all of the commands for all of the files in a project, there are many independent sets of operations that produce files

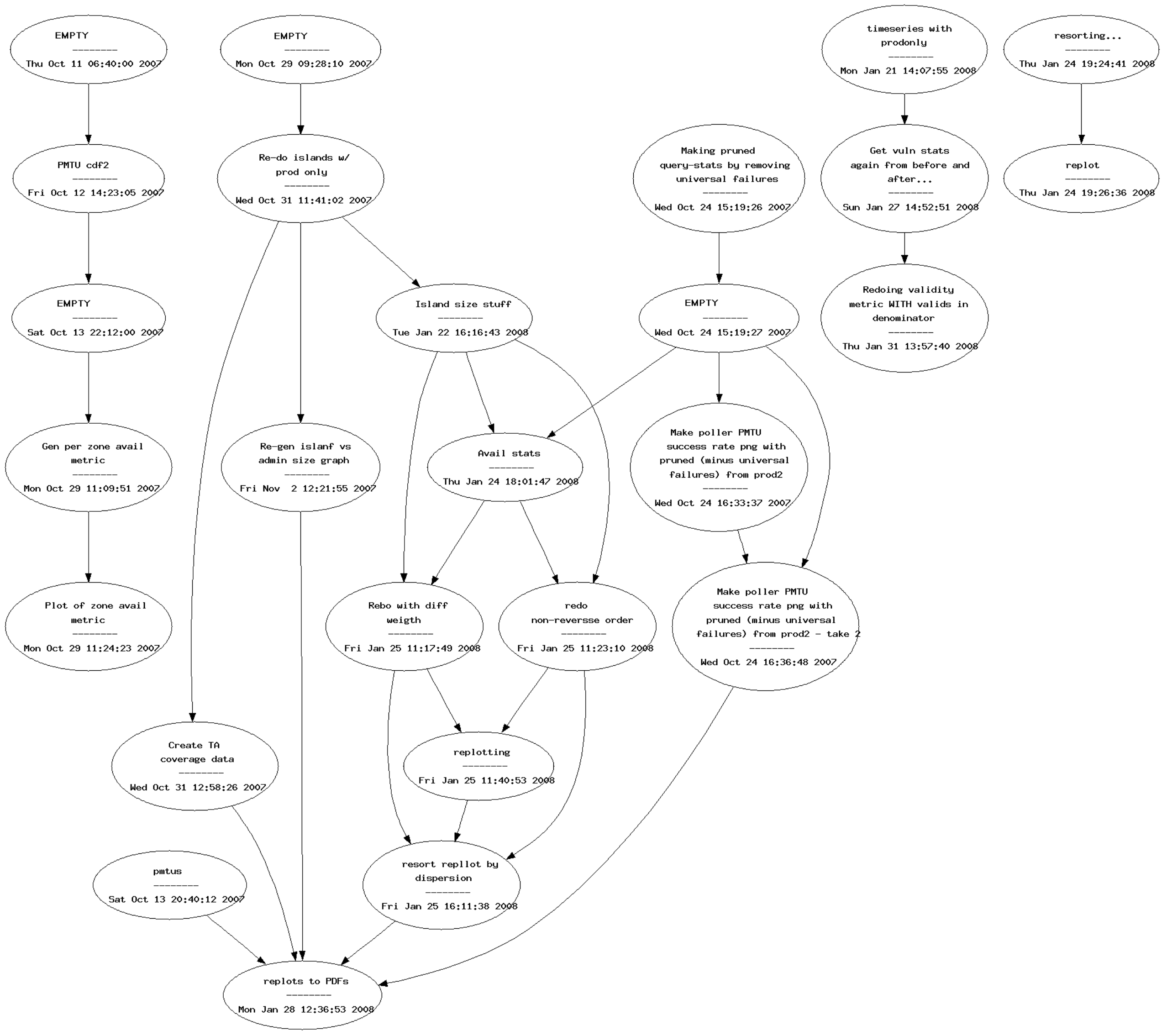


Figure 4: The provenance relationships for multiple result-sets is represented by experiments (nodes) and the edges between them (file dependencies). Textual descriptions were provided by the user at the time of experimentation.

that are unrelated to each other. As a result, when trying to recreate a single file, this user would waste quite a bit of wall-clock time, processing cycles, and potentially disk space if he were to simply re-execute his entire manual lab-book.

The general terseness of a manual lab-book may be unavoidable. This is simply because, the amount of extra meta-data needed to annotate commands and reproduce provenance is unrealistically high. It may often be the case that the researcher doesn't realize that their current processing depends on some files, and copy-and-pasting commands does not capture file dependencies. This user experienced first hand that maintaining a manual lab-book means that he (the researcher) has to manually manage the complexity of command dependencies.

After creating a manual lab-book for a specific project, this user began addressing all of the familiar drawbacks mentioned above. However, this time the user began using `lbsh` to help clean-up his manual lab-book in an automated way. After manually determining a set of commands that were needed for each of his graphs, he used `lbsh` to start a new experiment for each graph and logged all of the relevant commands (and implicitly logged all of the meta-data too). This made a lab-book entry for each logical result-set. Afterwards, this user had effectively organized his manual lab-book into a set of `lbsh` experiments.

Though this is not a complete adoption of `lbsh`'s environment, and there are further ways in which it could be used, it is very useful to see that as a general tool it does not prescribe a specific usage model which users must follow. In this case, the user was able to use part of what `lbsh` offered for a preexisting project and benefited from its organization. Furthermore, this case clearly shows that maintaining a manual lab-book can lead to a great deal of complexity and may (even under ideal situations) not be usable.

6. FUTURE WORK

`lbsh` has already helped a number of researchers, and it enjoys an active development effort. Recent interest has suggested that there are a few specific avenues for immediate improvement, and the beginnings of feature requests have begun to arrive.

Several independent users have suggested that optionally version-controlling a user-specified *source* directory would allow users to, for example, modify a configuration file before each experiment or make changes to processing scripts between experiments. Our near term plans are to integrate with the tool SubVersion [13]. `lbsh` could then capture the status of files in the source directory before each experiment. Users could then operate in a mode where not all needed information was in their environment or on the command line.

In addition, some users have suggested that being able to use the provenance of result-sets to determine which data files are no longer needed would allow researchers to cleanup disk space.

Other users have suggested that their preference for performing experiments is to use multiple shells concurrently. Users have requested a means to logically associate multiple experiments as this type of concurrent operation.

In these 2 cases, `lbsh` can already accommodate these features because of the meta-data it already stores. All that is needed is a front-end for post-processing in this way. Future work is to use this type of feedback to enrich the set of standard utilities that are already in `lbsh`.

7. DISCUSSION

Experimental results are a tool for researchers to convey their findings. By publishing results and providing data, we share what we have learned. However, more is required. Our goal should be to provide our findings in such a way that others can learn from them and in some cases build upon them. This is the hallmark of science.

Many examples exist of tools and approaches that help researchers document and formalize their experimentation so that results can be cleanly conveyed. However, very little evidence exists that these tools have gained mindshare among Computer Science researchers. We argue that a viable solution needs to afford researchers the flexibility that they already have in performing experiments in diverse environments, and must not add significant overhead to their existing practices. What is required is some form of diligence that documents the steps taken and any relevant input and meta-data used when researchers experiment.

We have seen that even the highly diligent approach of documenting all steps in experiments has serious drawbacks. We have also suggested that much of the diligence needed can be done by an automated program such as `lbsh`. We believe that `lbsh` has hit a sweet-spot by doing a lot of simple diligence while requiring essentially no overhead from and no retraining of researchers.

`lbsh` not only runs silently beside researchers, it also has very powerful provenance facilities to directly help researchers to post-process their own work, inspect each others work, and reproduce result-sets. In Section 5.1 we documented a case in which `lbsh` not only rediscovered how results were generated, it also reran them on new data and facilitated followon experimentation. In addition to this, in Section 5.2 a separate case demonstrated how `lbsh` excels over even the formal diligence of a manual lab-book, and that it does not prescribe any specific usage model. By logging commands into its automated lab-book, `lbsh` allows users to query their previous work and do general analysis by simply be invoking `lbsh`'s tools. Thus, users greatly reduce the overhead of managing their manual lab-books, and don't have to hunt through `.history` files and copy-and-paste commands manually.

It is important to note that `lbsh` is not fool-proof. Its method of scraping commands from the worker shell's output means that it does not record text that is not echoed. For example, it does not record passwords. Thus, while it doesn't risk exposing what it doesn't record, it cannot automatically re-run a command that needs a password to be input. Also, some processing (such as simulations) can add in time-varying elements (such as timestamps or randomness). Such experiments may produce different results every time they are rerun. In these cases `lbsh` cannot automatically reproduce results that are generated. However, another researcher can still inspect `lbsh`'s script files to understand the steps taken, and can inspect processing scripts and simulator code to understand their operation.

An example of the need for this type of facility could be the Internet Measurement Conference[1], which requires candidates for the best-paper award to make their datasets public. Perhaps augmenting this requirement with the provenance of the datasets would be even more helpful. In addition, the ACM SIGMOD[3] conference requests that authors include details and meta-data about their experimentation that makes the derivation of data understandable, and essentially equates to the type of data in a `lbsh`-style lab-book.

It is our hope that `lbsh` can find use throughout our community and can enable researchers to easily share their work with each other. Our observation is that sharing result-sets alone and including processing scripts is not always completely descriptive. In contrast, a tarball of a lab-book directory, initial datasets, an `.lbshrc` file, and any processing scripts would be sufficient as a way to convey provenance with result-sets. By encouraging `lbsh` archives as a sharing mechanism, our hope is that we may enjoy a simple and effective way to facilitate transparency into and reproducibility of result-sets.

8. REFERENCES

[1] Internet measurement conference. `http://www.imconf.net/`.

[2] Pound-shell. `http://lbsh.cs.ucla.edu/`.

[3] Special interest group on management of data. `http://www.sigmod.org/`.

[4] Writing the laboratory notebook. `http://home.clara.net/rod.beavon/lab_book.htm`.

[5] I. Altintas, B. Ludaescher, S. Klasky, and M. A. Vouk. Introduction to scientific workflow management and the kepler system. In *SC '06: Proceedings of the 2006 ACM/IEEE conference on Supercomputing*, page 205, New York, NY, USA, 2006. ACM.

[6] A. Barker and J. van Hemert. Scientific workflow: A survey and research directions. *The Third Grid Applications and Middleware Workshop (GAMW 2007)*, Sept. 2007.

[7] J. P. E. Fernandez. Programming python, part i. *Linux J.*, 2007(158):2, 2007.

[8] I. Foster, J. Voeckler, M. Wilde, and Y. Zhao. Chimera: A virtual data system for representing, querying, and automating data derivation. In *14th Conference on Scientific and Statistical Database Management, 2002.*, 2002.

[9] J. Frew and R. Bose. Earth system science workbench: A data management infrastructure for earth science products. In *SSDBM '01: Proceedings of the Thirteenth International Conference on Scientific and Statistical Database Management*, page 180, Washington, DC, USA, 2001. IEEE Computer Society.

[10] H. Kanare. *Writing the Laboratory Notebook.* American Chemical Society, May 1985.

[11] J. D. Myers, T. C. Allison, S. Bittner, B. Didier, M. Frenklach, J. William H. Green, Y.-L. Ho, J. Hewson, W. Koegler, C. Lansing, D. Leahy, M. Lee, R. Mccoy, M. Minkoff, S. Nijsure, G. V. Laszewski, D. Montoya, L. Oluwole, C. Pancerella, R. Pinzon, W. Pitz, L. A. Rahn, B. Ruscic, K. Schuchardt, E. Stephan, A. Wagner, T. Windus, and C. Yang. A collaborative informatics infrastructure for multi-scale science. *Cluster Computing*, 8(4):243–253, 2005.

[12] J. K. Pedersen. Linux gazette: Features of the tcsh shell. *Linux J.*, page 19.

[13] M. Pilato. *Version Control With Subversion.* O'Reilly & Associates, Inc., Sebastopol, CA, USA, 2004.

[14] R Development Core Team. *R: A Language and Environment for Statistical Computing.* R Foundation for Statistical Computing, Vienna, Austria, 2008. ISBN 3-900051-07-0.

[15] C. Ramey. What's gnu: Bash-the gnu shell. *Linux J.*, page 13.

[16] M. Richardson. Larry wall, the guru of perl. *Linux J.*, page 2.

[17] J. G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library User Guide and Reference Manual (With CD-ROM).* Addison-Wesley Professional, December 2001.

[18] Y. L. Simmhan, B. Plale, and D. Gannon. A survey of data provenance in e-science. *SIGMOD Rec.*, 34(3):31–36, 2005.

[19] J. Strang, L. Mui, and T. O'Reilly. `termcap` *&* `terminfo`. Third edition, Apr. 1988.

[20] J. Yu and R. Buyya. A taxonomy of scientific workflow systems for grid computing. *SIGMOD Rec.*, 34(3):44–49, 2005.

[21] J. Zhao, C. Goble, M. Greenwood, C. Wroe, and R. Stevens. Annotating, linking and browsing provenance logs for e-science. In *First International IFIP Conference on Semantics of a Networked World: ICSNW 2004*, 2004.