Improving Reproducibility of Distributed Computational Experiments

Quan Pham*
Department of Computer Science,
The University of Chicago
Chicago, IL
quanpt@cs.uchicago.edu

Tanu Malik, Dai Hai Ton That, Andrew Youngdahl
School of Computing, DePaul University, Chicago, IL, 60604, USA
{tmalik1,dtonthat,ayoungdah}@depaul.edu

ABSTRACT

Conference and journal publications increasingly require experiments associated with a submitted article to be repeatable. Authors comply to this requirement by sharing all associated digital artifacts, i.e., code, data, and environment configuration scripts. To ease aggregation of the digital artifacts, several tools have recently emerged that automate the aggregation of digital artifacts by auditing an experiment execution and building a portable container of code, data, and environment. However, current tools only package non-distributed computational experiments. Distributed computational experiments must either be packaged manually or supplemented with sufficient documentation.

In this paper, we outline the reproducibility requirements of distributed experiments using a distributed computational science experiment involving use of message-passing interface (MPI), and propose a general method for auditing and repeating distributed experiments. Using Sciunit we show how this method can be implemented. We validate our method with initial experiments showing application re-execution runtime can be improved by 63% with a trade-off of longer run-time on initial audit execution.

KEYWORDS

Network provenance, Record and replay, Sciunit, reproducibility of distributed objects,

ACM Reference Format:
Quan Pham and Tanu Malik, Dai Hai Ton That, Andrew Youngdahl. 2018. Improving Reproducibility of Distributed Computational Experiments. In *Proceedings of Submitted to First International Workshop on Practical Reproducible Evaluation of Computer Systems. (P-RECS'18).* ACM, New York, NY, USA, Article X, 6 pages. https://doi.org/XX.XXX/XXX_X

1 INTRODUCTION

The scientific publication has been central to the practice of science since the 17^{th} Century. It provides a means to describe the scientific method, make scientific claims, and present new results to validate the claims. Over the centuries the form of publication has remained largely stagnant while the methods used in science have changed dramatically. Almost all scientific experiments now involve at least some amount, if not a large amount of computation, to run simulations and/or to extract, analyze, and to store information. The static text-based form of publication is seriously insufficient for the current scientific method, be it helping the reader to grasp descriptions, follow logic, or interpret the results.

Recently several tools and standards have emerged which guide authors to aggregate digital artifacts associated with a computational experiment (e.g., Popper [11]) or automate the aggregation process (e.g., ReproZip [6], Sciunit [14]). When automating, the tools primarily use application virtualization (AV) [10]. In AV, operating system calls are interrupted to copy binaries, data, external user input, software dependencies, and associated provenance into a container. For instance, in Sciunit, when the container is re-run in a different environment, references to binaries and files are redirected to archived locations within the container thereby isolating it from the new host environment. Provenance associated with the experiment helps to verify if the new execution is an exact repetition of the previous execution [2, 18].

These tools make computations portable and verifiable across environments and are increasingly used to automate sharing and reproducing of computational experiments. The automation, however, is currently limited to containerizing and repeating local or non-distributed experiments. Distributed experiments such as simulations conducted by executing parallel programs or experiments involving client-server interaction cannot be currently reproduced using these tools, limiting usability.

A simple way to extend the tools to distributed experiments is by auditing network system calls during application virtualization. AV methods such as *ptrace* can be used to package associated network communication to be replayed later. While simply packaging network communication is sufficient for replay on the same number of resources, for computational reproducibility the packaged network communication must be replay-able on a variable number of available resources. Consider a user who has recently published an improved distributed k-means algorithm, and has published a paper claiming horizontal scalability of the distributed algorithm as the number of resources are increased and linear scalability as the size of dataset increases. While a reviewer would like to verify the scalability claims of the author, it is quite likely that the reviewer does not have the same environment setup or even the number of resources available to test the experiment as the author.

Toward this goal, in this paper we present a *network-enabled* Sciunit that allows researchers to capture and then replay distributed computational experiments. We use *ptrace* to audit network systems calls such that audited network communication can be replayed

*Work done when the author was a Ph.D. student at The University of Chicago

P-RECS'18, June 11, 2018, City, State Country

ACM ISBN XXX-XXXX-XX-XXX/XX/XX…$15.00
https://doi.org/XX.XXX/XXX_X

without the need of other network-connected processes. All network communication from this subsequent execution is emulated via the data captured from a previous execution. Hence, we can re-execute all computation in a single node without the need of other computing or storage nodes from the original experiment setup. Such a mode of re-execution is crucial if distributed experiments need to reviewed without access to large-scale network infrastructure. However, arbitrary scaling during replay is still part of our ongoing work. We show how a network-enabled *ptrace* can be used to audit a distributed experiment with minimal setup, without the need for system level privileges, and without changes to the original experiment's binaries or runtime environment. Finally, we show how provenance is recorded during distributed execution and merged back into a single database to allow for offline replay and subsequent verification. Our current network-enabled Sciunit can also be downloaded [8].

The remainder of the paper is structured as follows. We provide a description of the local application virtualization used by current reproducible tools in Section 2. We describe audit and replay of a distributed experiment in Sections 3 and 4 respectively. in Section 5 we describe the process of capturing the network provenance of a computational experiment and how the distributed provenance records are merged. We describe our evaluation in Section 6. Section 7 provides an overview of the related work in this area. Finally, the conclusion and future work are presented in Section 8.

2 LOCAL APPLICATION VIRTUALIZATION

Local application virtualization is described in detail in [14]. We describe it briefly again for review. The local application virtualization consists of two modes: an audit mode to create a container, and an execution mode to re-run a container [13]. In AV audit mode, a container of a user application is created as the user executes the application (in the context of auditing, such an execution is termed a *reference execution*). During execution, the Linux *strace* utility is used to monitor the running application process. *Strace* internally attaches itself to the process using the *ptrace* system call to monitor all the system calls of the running process. It intercepts non-network based POSIX system calls to determine the running process' state and the arguments to the system call. For example, when a process accesses a file or a library using the system call *fopen()*, the *fopen()* call is intercepted. The intercepted system call is "paused" to examine input arguments and the process control block. For instance, in *fopen()*, the file path parameter is extracted. By intercepting all calls, AV auditing determines all program binaries, libraries, scripts, and environment variables that a user program is dependent on.

The system call pause time is brief, requiring only two lightweight context switches added to the normal system call flow; experiments show that the overhead of intercepting system calls is minimal. During the pause, the identified dependencies are used in two ways: first, to create a "sandbox" application container that includes all identified dependencies, and second, to create an interaction log of the reference execution. The sandbox container is named with a package hash and placed in a special "root path", and contains all the dependencies that were identified during the reference execution audit. The dependencies are placed at the same path within the special root path as they were identified in the original system.

In AV execution mode, the application is executed from the container itself by monitoring its processes with *strace*, interrupting application system calls and extracting their path arguments, and redirecting all system call paths to paths within the special root path of the sandboxed container. By redirecting all application file requests into the container, the AV execution method fools the application program into believing that it is executing on the original audit-time machine with original file paths [13]. A provenance log generated during the AV audit phase contains interactions between processes when they are forked or execed, or between processes and files when files are opened or closed.

The *ptrace* mechanism, while sufficient for containerizing an application locally, cannot monitor a remote-spawned process. As an example, if a process uses SSH to create a new process on a remote machine, the new remote process is *fork*ed by SSH daemon on the remote machine. This breaks the ability of *ptrace* to audit that remote process.

3 DISTRIBUTED APPLICATION VIRTUALIZATION: AUDIT

Distributed application virtualization also works in audit and execution mode, but uses a network-enabled *ptrace* to monitor a remote spawned process. In particular, all remote processes are run within the context of this network enabled *ptrace*. To ensure that all remote processes are run within the context of network enabled *ptrace* we copy the network-enabled *ptrace* before the remote process is spawned and enforce that any remote process initiated during application virtualization is run within the network-enabled *ptrace*. Among different common process launchers, we select SCP and SSH to provide network-enabled *ptrace* copies on remote machines. In particular, when a program spawns a process on a remote machine, it invokes *execve(path-to-ssh, remote-machine, path-to-new-process, other-paramters, ...). ptrace* intercepts this *execve()* system call and performs four steps:

(1) ***Extract*** *remote host parameters from* execve() *system calls.* This involves going through *execve()* parameters and looking for the first parameter that is not an SSH option or an argument of an SSH option.
(2) ***Copy*** *network enabled* ptrace *to remote host.* To do this efficiently we first check if the network enabled *ptrace* binary is available on the remote host. If it is not, network-enabled *ptrace* is injected into the remote host.
(3) ***Inject*** *network enabled* ptrace *into* execve() *so that it executes before the remote process* The new parameters of *execve()* are *execve(path-to-ssh, remote-machine, path-to-PTU, path-to-new-process, other-paramters, ...).* Then let the *execve()* system call execute (Figure 1).
(4) ***Retrieve*** *provenance records from the remote machine.* Recorded provenance from all remote processes involved in the experiment are merged back to the root machine once the remote execution completes. This operation is further discussed in detail in Section 5.

Alternatively if the environment is more managed, the network-enabled *ptrace* can be made available within a common shared

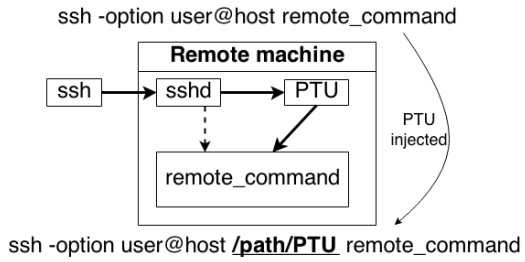

Figure 1: SSH injection

directory among all machines, e.g. network-shared users' home directory *$HOME*.

As part of distributed application virtualization, network communication between a local and a remote process is captured. We define *network communication* as data transmission from one process to another using a socket interface and without involving permanent files in a local file system, network file system, or shared memory. Network communication starts by having two or more processes use sockets for data transfer. On one machine a process listens on an agreed port number. Another process connects to that port number to initialize a network connection channel. These processes then exchange data by reading and writing to this socket.

There are two parts of this network audit: **meta audit**, in which connection metadata is audited, and **content audit**, in which actual data transferred in network connections is audited.

Meta Audit In meta audit, we monitor four system calls: *bind()*, *listen()*, *accept()*, and *connect()*. However, returned results from these system calls do not contain enough information about the source and destination network endpoints. To address this we record additional meta data from the */proc/net/* system directory. Since we use *ptrace* to interrupt network system calls we can delay the close of a socket until we finish recording the wanted network meta data.

Content Audit To support network replay, we intercept all the network calls and capture network data transferred in each socket connection. Parameters and memory buffers from network system calls are recorded in a provenance database as specified in Table 1.

To facilitate searching for recorded network system calls, network content audit implements a numbering scheme based on a *time order* of the system calls. We use a per-component numbering scheme in addition to recorded time to identify corresponding connections for later re-execution. Due to the nature of UNIX network system call API, data sending and receiving system calls are enumerated per sockets; connection initialization system calls are enumerated per processes.

The distributed application virtualization creates a container per machine which is copied on the master as network connections close. Thus the entire distributed container is available on the master machine after the experiment finishes.

4 DISTRIBUTED APPLICATION VIRTUALIZATION: REPLAY

In this section we discuss both re-execution and replay of an experiment. We use the term "re-execution" to mean the repeat of a distributed computation with each remote process executed again in its entirety. We use the term "replay" to mean a repeat of all computation at the original root node, but with the responses of the original remote nodes supplied through the content data captured during the original audit of the experiment. The former can be provided either on the original compute cluster, or on a different cluster with an identical number of nodes by supplying the new hostnames or IP addresses. The latter is provided if a repeat of the local processing is desired, but no suitable remote nodes are available.

System Calls	Recorded/Injected Values
listen, connect write, send sendto, sendmsg	**returned value**
accept	**value** = int accept(int sockfd, struct sockaddr *__**addr**__, socklen_t *__**addrlen**__);
read	**value** = ssize_t read(int fd, void *__**buf**__, size_t count);
recv	**value** = ssize_t recv(int sockfd, void *__**buf**__, size_t len, int flags);
recvfrom	**value** = ssize_t recvfrom(int sockfd, void *__**buf**__, size_t len, int flags, struct sockaddr *__**src_addr**__, socklen_t *__**addrlen**__);
recvmsg	**value** = ssize_t recvmsg(int sockfd, struct msghdr *__**msg**__, int flags);

Table 1: Network system call audit and replay: bold parameters and bold returned values are recorded during audit and injected back for replay

To perform a repeat, *ptrace* is again employed to intercept network system calls and then to locate which audited values to replace into the intercepted calls using both the numbering scheme mentioned in Section 3 and the three following steps:

- Match a current executing process to an audited process from the database;
- Match a socket to an audited socket of that audited process; and
- Match a current network system call to an audited system call of that audited socket.

Once an audited system call is located, `Sciunit` retrieves the corresponding content data from its database and replays the audited values in lieu of the network request. Audited parameters and data are inserted into the result buffers of the corresponding system call without actually invoking that call (Table 1).

Network repeat and replay limitations. While many test cases show the success or our network replay functionality, there are still some limitations. Even though the contents of network communication can be exactly repeated, in some cases replaying the same response data to a network call will not guarantee process behavior identical to the original execution. We describe as follows some information that cannot be replayed or re-executed in our framework.

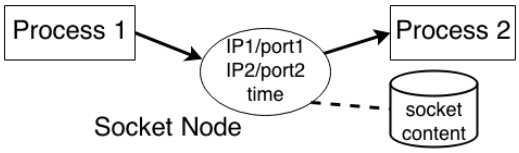

Figure 2: Socket Node stores meta information and transferred data

(1) Cryptography, especially secure communication in the presence of third parties, breaks network replayability of applications. In particular, in SSH [17], the cryptography authentication protocol using a couple of public-private key cannot be replayed.
(2) Data communications that use certain types of replay-attack countermeasures: session tokens, one-time passwords, and time stamping, are not replayable. Another observation is that DNS lookup queries and answers may be different for subsequent runs of the same process; hence DNS lookup is not network replayable.
(3) Network communications that send and receive control data instead of actual data content are not replayable. MPICH [9] passes file descriptors between processes via UNIX socket. While `Sciunit` can replay the communication, the passed file descriptors are only valid in the referenced execution and not valid in the re-execution.

5 PROVENANCE COLLECTION DURING DISTRIBUTED APPLICATION VIRTUALIZATION

In addition to labeling activities and entities as processes and files respectively, a network-data-to-process dependency is added to the provenance graph using a network vertex . We use a socket node as a specific data artifact that stores meta data about a network TCP/IP socket (i.e. source, destination IP address and port, connected time, etc.). However, to support network replay in later execution, actual data transferred by a socket also needs to be captured. Hence we define Socket Node to include actual data content of a network socket (see Figure 2). In this setup, the socket node contains both meta information and its content.

We also introduce a transparent network-process-to-process dependency. Distributed experiments are composed of many processes or tasks. Each process or task can be launched in an individual processing unit (or node) within the distributed platform. In UNIX environments, SSH [17] and MPICH [9] can be used to deliver the tasks to remote processing nodes. While dependencies between distributed processes are not as easily captured as dependencies between local resources, they are important nonetheless. The W3C PROV [16] has specifications for general process-to-process dependencies, but is not clear on how to represent that a process "wasTriggeredBy" a remote machine through an SSH connection. In our approach, we clarify that dependency by connecting the remote process directly to SSH with a "wasTriggeredBy" edge.

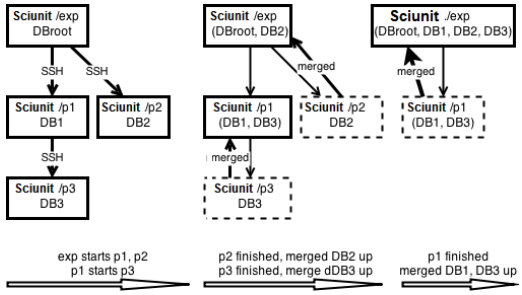

Figure 3: Merge local provenance databases

We describe the distributed databases used to keep provenance records when auditing, and how these distributed databases are merged back to the root machine.

We make the following assumptions:

(A1) Provenance audit should have minimal interference to application processes. Minimal network overhead should be incurred.
(A2) Computing nodes and their local disks may not be available for the entire experiment duration. Users may manually or automatically add or remove computing nodes based on computing demand. Once a computing node is idle, it might be taken offline to free allocated resources or to save budget. We will assume the node that starts the experiment is available for permanent storage.
(A3) No connectivity should be assumed among computing machines unless actual SSH connections exist. A simple scenario is that a user uses machine A to create computing jobs on machine B via SSH. Machine B is a login node of a separated cluster and jobs on B create tasks on machine C of that cluster via SSH. It is common for machine C to be behind a firewall that prevents machines from outside of the cluster to connect to it. Hence A cannot connect to C directly.

To satisfy requirement A1, provenance records are stored locally on each executing node during the auditing phase. A LevelDB database will be created on each node once an SSH connection is established to that node. If an experiment consists of multiple tasks across multiple nodes, each task's provenance will be recorded by a LevelDB on the task's node. A spanning tree is formed which connects the root node and all other nodes in the experiment.

Considering assumption A2, remote node LevelDB databases are moved to more permanent storage as soon as the remote process finishes. By the end of the experiment these distributed databases will be merged back to one single LevelDB database on the experiment's root node. Due to constraint A3 we merge each remote database to their spanning tree parent node, and repeat as each remote child process completes, until all processes have completed and all local databases are merged to the LevelDB database in the root node (Figure 3).

6 EXPERIMENT

We have implemented the distributed application virtualization within `Sciunit` [2]. In this section, we present our preliminary

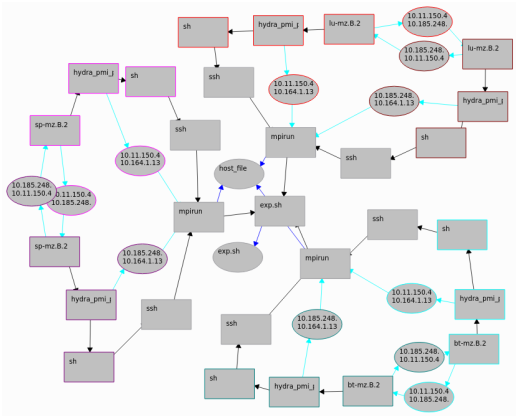

Figure 4: Provenance graph with Socket node and transparent SSH process dependency

experiment in using this distributed **Sciunit** to capture and replay an MPI experiment.

6.1 MPICH NASA Parallel Benchmark (NPB)

Figure 4 shows our evaluation of some test cases of The NAS Parallel Benchmarks (NPB) [4]. NPB is a set of programs designed for evaluating the performance of parallel systems. This benchmark contains five kernels (e.g., integer sort, random memory access; embarrassingly parallel; conjugate gradient and etc.) and three pseudo-applications *BT-MZ*, *LU-MZ*, *SP-MZ*. We selected NPB 3 and compiled this benchmark with three different applications *BT-MZ*, *LU-MZ*, *SP-MZ* and as parameter classes *A* and *B* (i.e, standard sizes), for 2 processors, and run on three Ubuntu workstations (64-bit, Intel Corei7 3.4Ghz, 8 GiB RAM). An instance with IP address 10.164.1.135 started MPI framework and spawned jobs to two slave instances with IP addresses 10.11.150.45 and 10.185.248.3. In this benchmark, MPICH used SSH to launch processes in other instances. In each of the two slave instances, SSH started *hydra_pmi_proxy*, which in turn started the benchmark binaries. The benchmark binaries communicated with others via network connections shown as socket nodes in the provenance graph Figure 4.

Table 2 shows overhead of using network-enabled *ptrace* for auditing provenance and network content. In both classes *A* and *B* benchmarks, there are 190 network system calls for each test case and the overall overhead is not significant with meta audit introduces 1-2% increase in runtime. Content audit shows slightly higher overhead than meta edit at 3-5% increase. This can be explained as more data were audited (i.e., ~524KB and ~268KB of data were captured for classes *A* and *B* respectively) and not only the meta data of network connections but also their actual transferring data was recorded.

6.2 RDCEP experiment

We also evaluated our system with another use case from the RDCEP paper [5]. In this test, its first step is to retrieve data from external websites. `Sciunit` has successfully re-executed the experiment offline without any access to those external websites. As shown in table 3, in the data retrieval step, meta audit introduces

	Normal	#Calls	Meta Audit	Content Audit
NPB BT-MZ.A.2	20.30			
NPB LU-MZ.A.2	15.74	190	~ 2.1%↑	~ 5.3%↑
NPB SP-MZ.A.2	14.84			
NPB BT-MZ.B.2	83.95			
NPB LU-MZ.B.2	71.02	190	~ 0.8%↑	~ 3.2%↑
NPB SP-MZ.B.2	59.12			

Table 2: NASA Parallel Benchmark runtime (seconds) and overhead of `Sciunit` meta audited mode (for query) and content audited mode (for replay) compared to normal NPB execution (no `Sciunit`)

	Normal	Meta Audit	Content Audit	Replay
Data retrieval	146.5±1.8	0.2%↑	134.5%↑	53.0±3.0
Other steps	varies	2% - 30% ↑ overhead		

Table 3: `Sciunit` performance (seconds) on network related and non-network related tasks from multi step experiment [5] shows 3-fold reduction in re-execution time with network replay.

almost no overhead, while content audit overhead is 134.5% due to extra time `Sciunit` spent for recording actual network data. Replay time for this experiment step is only 36% of the original execution duration. This 3-fold speedup is due to the data retrieval step loading data from the the local content-audited database instead of from remote servers. Overall, depending on users' requirements, using `Sciunit` meta audit or content audit will provide reasonable performance compared to normal execution.

7 RELATED WORK

TCPDUMP [1] is a tool to collect and replay network packets at the software level. TCPDUMP uses *ptrace* to capture system socket creation calls from applications and can retrieve information from the */proc* directory as well as the traffic across the socket. Tcpreplay [15] is a suite which gives the ability to use previously captured traffic from TCPDUMP. It allows a user to classify traffic as client or server, rewrite different layer headers, and also to replay the traffic. These tools provide support for network replay, but require applications to be implemented using their specific libraries.

Tools for building containers such as ReproZip [6] and Sciunit [14] do not address distributed computational experiments. Standards such as Popper [11] do not build containers but do provide guidance using several Unix utilities. Common Workflow Language (CWL) [3] descriptions are often run in a distributed manner where in the details of this distribution is up to the workflow executor and is not part of the CWL specification. CWL descriptions can also reference Docker software containers [7]. In this paper, we have focused how to automatically build a container from distributed computation experiments. Our technique is general, i.e, it enhances

ptrace and thus can be used by both Reprozip and Sciunit. Currently, it is far from a standard to be advocated; Further work is also needed to determine how automated containers (network-enabled or not) built either from Reprozip or Sciunit can be interfaced with CWL, which is increasingly being adopted as a descriptive standard.

Distributed provenance tracking is also described in SPADE [12]. However, in SPADE the focus is only of auditing provenance metadata and not content-audit. It also differs in terms of capturing network communication. In particular, SPADE employs polling operations on the results of *lsof* which can miss some short-lived connections. We have used *strace* to stop sockets from being closed until provenance is captured thereby eliminating the possibility of missing short lived sockets. Also, by capturing socket content we go further by allowing for replay of distributed experiments when the remote nodes are no longer available.

8 CONCLUSIONS AND DISCUSSION

In this paper we have described the limitations of current tools to enable reproducible analyses of distributed computational experiments. We have described how local application virtualization used in the current tools can be transparently extended to audit distributed experiments, and then replayed on a single node. Our method is independent of any workflow system, works at the user level of UNIX operating systems, and thus can be adopted by most researchers. Experiments show that the tool is efficient with only a low level of overhead introduced and works effectively with individual applications and computation frameworks.

Our goal is to address the problem of reproducibility across totally different distributed platforms. This implies repeating experiments across heterogenous nodes without any installation, but also take advantage of increased capacity or scale down to lesser endowed environments. This will require an extra-step to examine every single sub-process within the audited experiment and modify invocation of running these sub-processes while guaranteeing the correctness of the outputs. This is part of our ongoing work.

REFERENCES

[1] TCPDUMP/LIBPCAP public repository. URL http://www.tcpdump.org/.

[2] The Sciunit. https://sciunit.run/, 2017. [Online; accessed 10-Sep-2017].

[3] Peter Amstutz, Michael R. Crusoe, Nebojša Tijanic, Brad Chapman, John Chilton, Michael Heuer, Andrey Kartashov, Dan Leehr, Hervé Ménager, Maya Nedeljkovich, Matt Scales, Stian Soiland-Reyes, and Luka Stojanovic. Common Workflow Language, v1.0. 7 2016. doi: 10.6084/m9.figshare.3115156.v2.

[4] David H Bailey, Eric Barszcz, John T Barton, David S Browning, Russell L Carter, Leonardo Dagum, Rod A Fatoohi, Paul O Frederickson, Thomas A Lasinski, Rob S Schreiber, et al. The NAS parallel benchmarks. *International Journal of High Performance Computing Applications*, 5(3):63–73, 1991.

[5] N. Best, J. Elliott, and I. Foster. Synthesis of a complete land Use/Land cover dataset for the conterminous united states. *SSRN eLibrary*, 2012.

[6] Fernando Chirigati, Rémi Rampin, Dennis Shasha, and Juliana Freire. ReproZip: Computational reproducibility with ease. In *SIGMOD'16*, pages 2085–2088, 2016.

[7] CWL. Common Workflow Language Documentation, 2018. URL https://www.commonwl.org/draft-3/UserGuide.html.

[8] DePaulDBGroup. Network-enabled Sciunit, 2018. URL https://bitbucket.org/depauldbgroup/provenance-to-use/branch/network.

[9] William Gropp. MPICH2: a new start for MPI implementations. In *Recent Advances in Parallel Virtual Machine and Message Passing Interface*, pages 7–7. Springer, 2002.

[10] Philip J. Guo and Dawson Engler. CDE: Using system call interposition to automatically create portable software packages. In *USENIX*, 2011.

[11] Ivo Jimenez, Michael Sevilla, Noah Watkins, Carlos Maltzahn, Jay Lofstead, Kathryn Mohror, Andrea Arpaci-Dusseau, and Remzi Arpaci-Dusseau. The Popper convention: Making reproducible systems evaluation practical. In *Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, pages 1561–1570. IEEE, 2017.

[12] Tanu Malik, Ashish Gehani, Dawood Tariq, and Fareed Zaffar. Sketching distributed data provenance. *Data Provenance and Data Management in eScience*, 426:85–107, 2013.

[13] Quan Pham, Tanu Malik, and Ian Foster. Using provenance for repeatability. In *TaPP*, 2013.

[14] Dai Hai Ton That, Gabriel Fils, Zhihao Yuan, and Tanu Malik. Sciunits: Reusable research objects. In *IEEE eScience*, Auckland, New Zealand, 2017.

[15] Aaron Turner and Fred Klassen. TCPreplay - PCAP editing and replaying utilities. URL http://tcpreplay.appneta.com/.

[16] W3C. PROV-DM: The PROV data model, 2013. URL https://www.w3.org/TR/prov-dm/.

[17] Tatu Ylonen and Chris Lonvick. The secure shell (SSH) protocol architecture. 2006.

[18] Zhihao Yuan, Dai Hai Ton That, Siddhant Kothari, Gabriel Fils, and Tanu Malik. Utilizing provenance in reusable research objects. *Informatics*, 5(1), 2018. doi: 10.3390/informatics5010014.